Web程序员成功之路

MVC Web

开发学习实录

- 迅速提高读者Web开发能力，全面挖掘读者开发潜力
- 一线资深Web程序员经验力作，窗内网独家推荐自学教材
- 10个小时视频教学，简化学习过程
- 62个实战案例与理论知识综合讲解，提高应用能力
- 网站互动教学（www.itzcn.com），QQ群在线帮助读者解疑

杨晓军 秦 方 编著





清华大学出版社

Web 程序员成功之路

MVC Web 开发学习实录

杨晓军　秦　方　编著

清华大学出版社

北　京

内 容 简 介

本书共 15 章，围绕 Visual Studio 2010 下的 MVC 开发，首先介绍 MVC 环境的搭建及其与三层的区别，并创建了第一个 MVC 项目，然后重点对 MVC 中的路由、控制器、模型、视图、页面辅助类以及如何利用 WebFrom 控件和自定义视图引擎进行讲解，同时还包括过滤器、异常处理、整合 jQuery 和 Ajax，以及单元测试等高级课题。本书结构规范，讲解详略得体，选例典型、实用，学练结合，能有效提高学习成效。

本书适合具有一定 ASP.NET 基础和希望通过 MVC 提高技能的读者，同时也适合 ASP.NET MVC 的初学者学习，还可作为相关培训机构的教材及参考书。

图书在版编目(CIP)数据

MVC Web 开发学习实录/杨晓军，秦方编著. --北京：清华大学出版社，2011.7

(Web 程序员成功之路)

ISBN 978-7-302-25444-7

Ⅰ. ①M…　Ⅱ. ①杨…　②秦…　Ⅲ. ①网页制作工具—程序设计　Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字(2011)第 078423 号

责任编辑：张　瑜
装帧设计：杨玉兰
责任校对：周剑云
责任印制：

出版发行：清华大学出版社　　**地　址：**北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编：**100084
社 总 机：010-62770175　　**邮　购：**010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：190×260　　**印　张：**23.75　　**字　数：**570 千字
版　　次：2011 年 7 月第 1 版　　**印　次：**2011 年 7 月第 1 次印刷
附 DVD 1 张
印　　数：1~4000
定　　价：48.00 元

产品编号：

前　言

MVC 是一种设计模式，它把应用程序分成 3 个核心模块：模型、视图和控制器，它们各自处理自己的任务。在 Java 中的应用非常成熟，其他 Web 编程语言也有其对应的 MVC 框架，像 PHP 和 Ruby 等。

ASP.NET MVC 是微软今后开发 Web 应用程序的一个主流技术。Microsoft 推出的 Front Controller 式 Web 开发模型弥补了 ASP.NET 对 HTML 的控制能力不足，单元测试较为困难等缺点。一经推出，就受到广大程序员的欢迎。目前的最新正式版本为 ASP.NET MVC 2.0，也是本书中采用的版本。

本书内容

第 1 章　从头开始学 ASP.NET MVC 框架。本章介绍在 Visual Studio 2010 中创建 ASP.NET MVC 应用程序和单元测试项目的方法，讲解了关于 MVC 的一些重要概念，帮助读者快速入门，例如什么是 MVC、MVC 与三层的区别以及 ASP.NET MVC 的简介等。

第 2 章　畅通无阻——管理 URLRouting。本章我们来学习 URLRouting 的配置规则，了解 URLRouting 和 URL 重写的区别，然后学习使用工具调试 URLRouting。

第 3 章　Controller 及 Action。本章介绍 ASP.NET MVC 中控制器和动作，然后使用 Response.Write()方法在 Action 中直接输出一个 Web 页面，了解 Action 的返回值，以及带参数 Action 的用法，最后了解使用 RedirectToAction()方法执行动作之间的跳转。

第 4 章　Model。本章主要介绍 Model 在 MVC 中所扮演的角色以及 Model 的重要性。

第 5 章　简单实现绚丽的界面。本章主要介绍什么是 View 以及 View 在 MVC 中的地位，同时讲解 Controller 向页面视图中传递数据的方法和强类型的数据传递方式。

第 6 章　页面辅助类。掌握页面辅助类 HtmlHelper 的使用方法和 URL 辅助类的使用方法。

第 7 章　在 View 中使用 WebForm 控件。本章先介绍 ASP.NET MVC 默认视图的不足，然后教读者如何使用 WebForm 服务器端控件解决这个问题，并讲解在 MVC 视图中使用 Repeater 和 DataList 控件的用法。

第 8 章　自定义视图引擎。本章介绍 ASP.NET MVC 应用程序生成视图的原理，然后引入一个 StringTemplate 模板引擎来帮助我们生成视图，最后使用 ASP.NET MVC 提供的接口自定义一个视图引擎。

第 9 章　过滤器。本章介绍应用于 Action 或 Controller 的过滤器的设置方法，然后讲解缓存过滤器、异常过滤器等系统提供的几种过滤器。

第 10 章　MVC 异常处理技巧。本章介绍全局异常处理的方法，讲解如何使用异常过滤器来处理应用程序异常，最后简单叙述路由异常和动作异常的处理方法。

第 11 章　MVC 中 jQuery 的应用。本章介绍 JavaScript 框架 jQuery 的基本用法，其中包括 jQuery 的选择器和过滤器，以及对页面元素的搜索，掌握 jQuery 表单、CSS 等的操作，另外讲解了一些高级应用，比如 jQuery 动画的用法，日历控件的用法和对话框控件的用法等。

第 12 章　注入 Ajax 特性的 MV。本章介绍 ASP.NET MVC 中 Ajax 的使用方法。先简单了解使用手工编写 JavaScript 的方式实现 Ajax 功能，然后使用 jQuery 提供的一些方法方便地

实现 Ajax 的操作。

第 13 章　单元测试。本章介绍单元测试的意义和测试驱动开发(TDD)的思想，然后使用 Visual Studio 内置的测试功能对 MVC 中的 URLRouting 进行单元测试，最后学习对 Controller 中的 Action 进行单元测试。

第 14 章　MVC 博客系统。本章介绍采用 MVC 开发博客管理系统的整个过程，包含系统分析、数据库设计、系统设计、文章浏览、用户登录以及添加文章栏目等功能。

第 15 章　通讯录系统。本章介绍通讯录管理系统的功能模块设计和数据库设计方案，以及通讯录中用户、照片、权限和留言模块的实现。

本书特色

本书的大量内容来自真实 ASP.NET MVC 项目，力求通过读者实际操作时的问答方式，使读者更容易掌握 MVC 的管理和操作。本书难度适中，内容由浅入深，实用性强，覆盖面广，条理清晰。

(1) 结构独特。通过“网络教学→基础知识→实例描述→实例应用→运行结果→实例分析”形式将每个知识点与实际应用中的问题相结合。

(2) 形式新颖。用准确的语言总结概念，用直观的图示演示过程，用详细的注释解释代码，用形象的比方帮助记忆。

(3) 技术文档。将一些非常简单的知识点或者理论性的内容安排在这里，通常这些文档让读者了解有关的概念和术语。

(4) 内容丰富。涵盖了实际开发中 ASP.NET 技术所遇到的 URLRouting、Controller、View、过滤器、视图引擎、jQuery、Ajax 和单元测试等方面的热点问题。

(5) 随书光盘。本书为实例配备了视频教学文件，读者可以通过视频文件更加直观地学习 ASP.NET MVC 的知识。

(6) 网站技术支持。读者在学习或者工作的过程中，如果遇到实际问题，可以直接登录 www.itzcn.com 与我们取得联系，作者会在第一时间给予帮助。

(7) 贴心的提示。为了便于读者阅读，书中穿插一些技巧、提示等小贴士，体例约定如下。

- 提示：通常是一些贴心的提醒，加深印象或提供建议，以及解决问题的方法。
- 注意：提出学习过程中需要特别注意的一些知识点和内容，或者相关信息。
- 技巧：通过简短的文字，指出应用知识的小窍门。

读者对象

本书具有知识全面、实例精彩及指导性强的特点，力求以全面的知识点及丰富的实例来指导读者透彻地学习 ASP.NET MVC 网络软件开发。本书可以作为 ASP.NET MVC 的入门书籍，也可以帮助中级读者提高技能，对高级读者也有一定的启发意义。

本书适合以下人员阅读学习：

- ASP.NET MVC 初学者以及在校学生。
- 网站开发人员。
- 网站维护人员。
- 网页制作爱好者。
- 各大、中专院校的在校学生和相关授课老师。
- 其他 ASP.NET 从业人员。

作者团队

本书主要由杨晓军、秦方编写，其他参与编写、资料整理、程序开发的人员还有张凡、周衡坤、郭旭辉、刘小丹、王咏梅、孙晓芳、郝春雨、卢江等。

由于编者水平有限，书中难免存在不足和疏漏之处，恳请读者批评指正。

目　　录



第 1 章　从头开始学 ASP.NET MVC 框架

内容摘要

MVC 模式的概念很早就提出来了，而且在 Java 中的应用非常成熟，其他 Web 编程语言也有其对应的 MVC 框架，像 PHP 和 Ruby 等。

相比较而言，ASP.NET 的起步比较晚，算是后起之秀，它解决了传统 ASP.NET WebForm 编程具有的高度封装和制作高性能网站的效率瓶颈。一经推出，便受到广大程序员的欢迎。

目前最新正式版本为 ASP.NET MVC 2.0，也是本书采用的版本。通过本章的学习，读者可以在 Visual Studio 2010 中轻松地创建 ASP.NET MVC 应用程序，查看目录结构和 MVC 路由，以及为其添加单元测试项目。

此外，本章还准备了一些基础知识，对学习 MVC 时的重要概念进行讲解，帮助读者快速入门。像什么是 MVC，MVC 与三层的区别以及 ASP.NET MVC 的简介等。

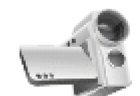
学习目标

- 了解什么是 MVC 及其组成部分
- 了解 MVC 与 ASP.NET MVC 的关系
- 了解 ASP.NET MVC 与 WebForm 的区别
- 掌握 ASP.NET MVC 应用程序的创建过程
- 熟悉 ASP.NET MVC 应用程序的命名空间以及目录结构
- 了解 ASP.NET MVC 路由的作用
- 了解 ASP.NET MVC 应用程序中模型、视图和控制器的定义
- 熟悉 ASP.NET MVC 应用程序单元测试的方法

1.1　MVC 与三层架构之间的抉择

在开始 ASP.NET MVC 2.0 的学习之前，我们得先弄明白什么是 MVC？

简单地说，MVC 就是 Model“View”Controller 的缩写，中文直译意思为“模型、视图、控制器”。

视频教学：光盘/videos/01/1.1　MVC 简介　　长度：10 分钟

1.1.1　基础知识——MVC 简介

抛开具体的开发语言来讲，MVC 是一种软件开发的架构模式，也算是一种思想。在 MVC 模式中将软件系统分为模型、视图和控制器三个基本部分。

它们各司其职，相互独立，又具有联系，图 1-1 是 MVC 三部分的示意图。

- 模型　负责对整个软件项目数据和业务的封装和管理。
- 视图　负责给用户传递信息，并收集用户提交的信息。
- 控制器　负责控制视图的展示逻辑。

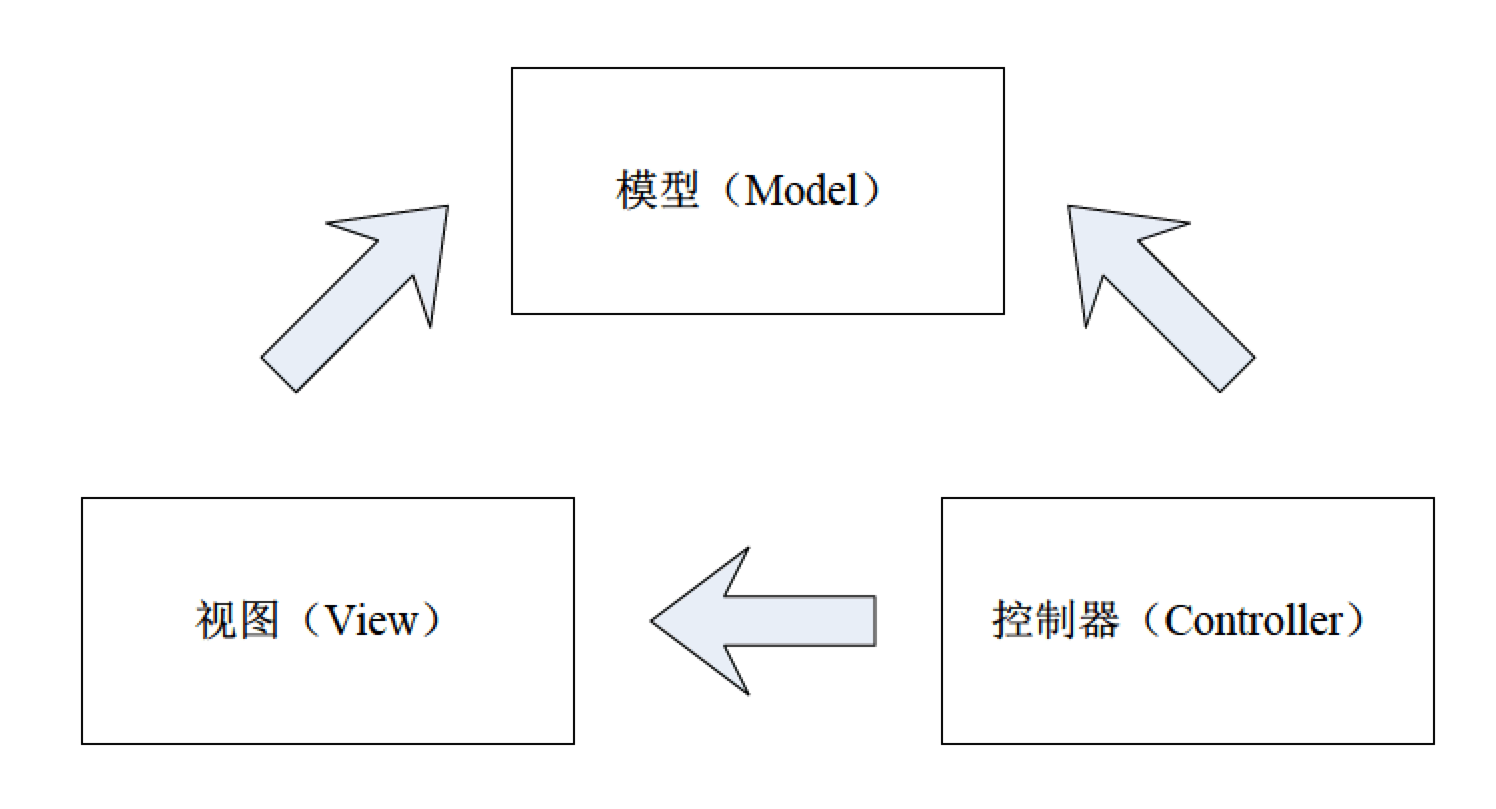


图 1-1　MVC 示意图

1. MVC 工作模式

1) 视图

视图是用户看到并与之交互的界面。对老式的 Web 应用程序来说，视图就是由 HTML 元素组成的界面，在新式的 Web 应用程序中，HTML 依旧在视图中扮演着重要的角色。但一些新的技术层出不穷，它们包括 Flash、XHTML、XML/XSL 和 WML 等标识语言和 Web Services。

如何处理应用程序的界面变得越来越有挑战性。MVC 的一大好处就是它能为你的应用程序处理很多不同的视图。在视图中其实没有真正的处理发生，不管这些数据是联机存储的还是一个雇员列表，作为视图来讲，它只是作为一种输出数据并允许用户操纵的方式。

2) 模型

模型表示企业数据和业务规则。在 MVC 的三个部分中，模型拥有最多的处理任务。例如，它可能用 EJB 这样的组件对象来处理数据库。被模型返回的数据是中立的，也就是说模型与数据格式无关，这样一个模型能为多个视图提供数据。由于应用于模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性。

3) 控制器

控制器接受用户的输入并调用模型和视图去完成用户的需求。单击 Web 页面中的超链接和发送 HTML 表单时，控制器本身不输出任何东西和做任何处理。它只是接收请求并决定调用哪个模型组件去处理请求，然后确定用哪个视图来显示模型处理返回的数据。

2. MVC 优点

在使用 ASP 或者 PHP 开发 Web 应用时，初始的开发模板就是混合层的数据编程。例如，直接向数据库发送请求并用 HTML 显示，开发速度往往比较快。但由于数据页面的分离不是很直接，因而很难体现出业务模型的样子或者模型的重用性，也很难满足用户的变化性需求。

MVC 要求对应用分层，虽然要花费额外的工作，但产品的结构清晰，产品的应用通过模型可以更好地体现。

首先，最重要的是应该有多个视图对应一个模型的能力。在目前用户需求的快速变化下，可能有多种方式访问应用的要求。

例如，订单模型可能有本系统的订单，也有网上订单，或者其他系统的订单，但对于订单的处理都一样，也就是说订单的处理是一致的。按 MVC 设计模式，一个订单模型以及多个视图即可解决问题。这样减少了代码的复制，即减少了代码的维护量，一旦模型发生改变，也易于维护。

其次，由于模型返回的数据不带任何显示格式，因而这些模型也可被直接应用于接口。

再次，由于一个应用被分离为三层，因此有时改变其中的一层就能满足应用的改变。应用的业务流程或者业务规则的改变只需改动 MVC 的模型层即可。

控制层的概念也很有效，由于它把不同的模型和不同的视图组合在一起来完成不同的请求，因此控制层可以说是包含了用户请求权限的概念。

最后，它还有利于软件工程化管理。由于不同的层各司其职，每一层上不同的应用具有某些相同的特征，有利于通过工程化和工具化产生管理程序代码。

3. MVC 缺点

凡事都不是绝对的，MVC 亦是如此。MVC 也不是最先进、最优秀或者最好的选择，其缺点主要体现在以下几个方面。

1) 增加了系统结构和实现的复杂性

对于简单的界面，严格遵循 MVC，使模型、视图与控制器分离，会增加结构的复杂性，可能产生过多的更新操作，从而降低了运行效率。

2) 视图与控制器间过于紧密的连接

视图与控制器是相互分离的，但是又有紧密联系的部分，如果视图没有控制器的存在，那么它的应用是很有限的。反之亦然，这样就妨碍了它们的独立重用。

3) 视图对于模型数据的低效率访问

依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

目前，一般高级的界面工具或者构造器不支持 MVC 架构。改造这些工具以适应 MVC 需要和建立分离部分的代价是很高的，从而造成使用 MVC 的困难。

1.1.2　实例描述

MVC 与三层架构的区别是什么？

关于这个问题从来都不缺内容，可以从网上的博客、论坛再到面试题等各个地方的出处足以见得话题之多之广。

然而，回答也是仁者见仁，智者见智。从学术角度来说，无论是 MVC 还三层架构，都是为了解决实际软件开发过程中遇到问题的统筹方法，所以从它们所解决的问题的角度来比较是最为合适的。

1.1.3　实例应用

三层架构分为界面层、业务逻辑层和数据访问层。很多人就将 MVC 里的三个部分与三层架构等同起来，认为界面层等于 View，业务逻辑层等于 Controller，数据访问层等于 Model，但这是完全错误的。

MVC 三层架构主要表现为如下几点。

- MVC 设计模式解决的是页面代码、页面控制逻辑和数据耦合的问题。它属于界面层，例如，ASP.NET MVC 和 Struts 都是界面层框架。
- MVC 里的 Controller 负责对页面进行控制，像页面间跳转，显示逻辑等。三层架构里的业务逻辑主要是对业务实体数据的加工，把加工后的数据传给页面显示。
- MVC 里的 Model 只是数据实体，不具备什么增、删、改、查的功能，它接收的数据是从业务逻辑层处理好传过来的数据。而三层架构里的数据访问层具有增、删、改、查功能，直接对数据库操作，为业务逻辑提供数据支持。

1.1.4　实例分析



源码解析

通过本节的内容，读者需要弄清这样几点。

(1) MVC 是由模型、视图和控制器三部分组成的，它们负责的功能不同，是相互独立的。

(2) 如何协调和区分 MVC 每个部分的工作模式。

(3) MVC 不是神，认识将项目 MVC 化的优点与缺点。

(4) MVC 不等于三层架构，了解两者之间的差别。

1.2　MVC 的应用现状

MVC 时代来临了，但是一开始是不被很多人接受的。主要的原因可能是：大家不得不告别拖拉控件的至爽感受，回到貌似 ASP 的历史岁月。

所以，心有不甘是可以理解的，然而时代显然是进步的。我们虽然必须在 View 中进行很多 HTML 代码的工作，但是 MVC 为我们提供了可以堪称完美的方案。

现在，我们就来看看哪些成熟的技术中有 MVC 的身影。

**视频教学**：光盘/videos01/1.2　MVC 的应用现状　　　　长度：5 分钟

1.2.1　实例描述

随着语言的不断成熟以及网络服务器软件的不断发展，MVC 很快就发现其在 Web 应用程序体系结构中的用武之地。但是直到 2004 年 7 月，MVC 才开始大行其道。

后来出现了很多的网络开发架构，而 MVC 作为这个市场的新宠有很多厂商追捧。

理解 ASP.NET MVC 的最佳方式就是先简单了解一下其他一些选择。

1.2.2　实例应用

目前，在互联网上主流的应用都是使用 Java、PHP、Rails、Python 或者 ASP.NET 开发的，而且它们都在不同程度上支持 MVC 架构，下面就来认识一下这些技术。

1) Struts 和 Java

在 Java 领域，Struts 是使用最多的 MVC 架构。此外，还有其他一些架构(例如，Spring 和 JSP)，但是 Struts 无疑是最好的。

Struts 于 2001 年 6 月发布，后来 WebWorks 和 Struts 团体合并，并创建了 Struts 2。Struts 常常被认为是标准的架构，并且越来越多地与 Spring 架构一起使用。

2) Zend 架构和 PHP

尽管 Web 的很多技术都是在 PHP 上运行的，但是 PHP 应用程序似乎没有对 Web 有足够的重视。很多开发人员认为 PHP 鼓励面条式的代码，而且也不难找到用名为 index.php 的单个文件编写的整个 PHP 应用程序。

Zend 架构(Zend Framework，ZF)是一个用 PHP5 实现的开放源码的 Web 应用程序架构。在向 PHP 开发人员提供某种弹性的同时，PHP5 试图给 PHP 的应用程序开发带来一些正式的东西。因为 Zend 架构提倡的是按意愿使用(use-at-will)的哲学，所以不要求开发人员使用这些组件。

此外，ZF 在可行的时候使用了配置上的约定来实现前端控制器的模型。它包括自己的视图模板引擎，同时支持插入到其他可替换的视图中。

3) Ruby on Rails

Ruby on Rails(简称 Rails)是为 Web 创建的最受欢迎的 MVC 架构之一，它的工作基于如下

两个主要指导原则。

- 约定胜于配置　其主要思想是作为已经进行网络开发多年的开发人员，那么就一定会有一些同性的问题，可以让其成为架构的一部分。
- 不要重复工作或要保持 DRY　这用于集中应用程序和代码的逻辑。如果心中始终保持这一念头，那么就会发现自己编写的代码要少很多。

此外，Rails 还包括路由的概念并通过 config/routes.rb 将 URL 映射到适当的控制器上，例如下面的 Rails 命令。

```
map.connect ':controller/:action/:id'
```

将把任何 URL 通过这个通用的格式发送到适当的控制器动作方法中，并将 ID 作为一个参数传递下去。因此，www.itzcn.com/news/page/10 将被发送到 NewsController 的 Page 动作中，传递 id 参数为 10。

4) Django 和 Python

Django 使用了 Python 语言并注重整洁的设计，将注意力放在尽可能地进行自动化设计上并重视 DRY 原则。

与 Rails 一样，Django 还包括自动生成管理接口的能力，常常称作脚手架。它包括一个可插入的模板系统，允许通过继承来进行模板重用，从而尽可能地避免冗余。

在几乎所有主要的 MVC 网络架构中都能找到路由的概念，Django 也不例外。在 Django 中，路由是在一个名为 urls.py 的文件中声明的。Django 使用了正则表达式(Regular Expression)来映射 URL 和方法之间的关系。例如，下面的代码片段：

```
(r'^(?P<object_id>\d+)/news/page/$', 'store.news.view'),
```

以 www.itzcn.com/news/page/10 的格式将任何 URL 发送到名为 store.news.view 的 Django 函数中，并将值为 10 的参数 id 传递进来。

5) MonoRail

MonoRail 是第一个基于 ASP.NET 的模型–视图–控制器架构，它的开发者 Castle Project 是受到 Ruby on Rails 的灵感启发。

创建 MonoRail 的目的是解决在使用 WebForm 受挫时所带来的问题，简化标准的 WebForm 范例。MonoRail 包括一个前端控制器的体系结构，指向应用程序中的入口是控制器的一个实例。这与 Page 的实例不同，Page 的实例是指向 ASP.NET 的 WebForm 应用程序中的标准入口。

MonoRail 是一个非常具有弹性的.NET 应用程序架构，允许插入和使用各种组件。它具有很多可扩充的点，且描述性很强，默认情况下可以引入很多开放源代码的程序，例如 Castle 自己的 ActiveRecord、NHibernate for Models 和 log4net for logging 等。

1.2.3　实例分析



源码解析

本节我们将对主流开发语言中 MVC 的应用框架有一定认识。

很多年来，MVC 模式一直都是计算机科学中非常重要的体系结构模式。自从引入 MVC 之后，它被应用到很多架构中，像 Java 和 Python 等。

此外，在 Mac 和 Windows 操作系统内部也可以找到 MVC 的应用。

甚至于作为区分一种新语言是否成熟的标志，就是看是否有其对应的 MVC 框架。

1.3　ASP.NET WebForm 与 MVC 的争论

使用微软 Visual Studio 工具开发 Web 应用程序主要有两种方式：一种是常用的创建 ASP.NET WebForm，另一种就是本书着重介绍的 ASP.NET MVC。



视频教学：光盘/videos/01/1.3　ASP.NET MVC 概述　　长度：11 分钟

1.3.1　基础知识——ASP.NET MVC 概述

作为设计模式的王者，MVC 在众多领域都成为良好模型的代名词。从前在 ASP.NET 下我们只能依靠 Monorail 来实现 ASP.NET 下无控件的 MVC。

但是现在 ASP.NET 下的 MVC 已经成为现实。ASP.NET MVC 是微软官方推出的基于 ASP.NET 的 MVC 模式网站应用程序开发框架，网站地址为 http://www.asp.net/mvc。

ASP.NET MVC 的第一个版本是于 2009 年 3 月 17 日发布的 RTM 版本。可以说自推出以来，就一直受到广大程序员的欢迎。

在经历了漫长的试用(Preview)之后，终于在 2010 年 3 月 11 日发布了 ASP.NET MVC 2.0。它可以与.NET Framework 3.5 的 SP1 一起使用，而且 MVC 2.0 随 Visual Studio 2010 安装程序自动集成。另外，Visual Studio 2010 也是本书所采用的开发工具。



ASP.NET MVC 2.0 是对 1.0 的又一次重大更新，兼容 ASP.NET MVC 1.0。和以前一样，ASP.NET MVC 2.0 的源代码完全开放。另外，ASP.NET MVC 2.0 可以与 ASP.NET MVC 1.0 并存。也就是说在同一台机器上，有的程序可以用 ASP.NET MVC 1.0，有的程序用 ASP.NET MVC 2.0。

ASP.NET MVC 2.0 框架具有以下功能。

1) 分离任务，易测性和默认的测试驱动组件

所有 MVC 用到的组件都是基于接口并且可以被 mock 对象测试到。开发人员可以不必在 ASP.NET 进程中运行 Controller 就可以使用测试，使得测试更加快速和简捷。

2) 可扩展的简便框架

MVC 框架被设计用来更轻松地移植和定制功能。开发人员可以加入自己的视图引擎，URL 重写策略，重载 action 方法等。

3) 强大的 URL 重写机制

让开发人员更方便地建立容易理解和可搜索的 URL。URL 可以不包含任何文件扩展名，并且可以重写 URL 使其对搜索引擎更加友好。

4) 重用模块及组件

可以使用 ASP.NET 现有的页面标记、用户控件和模板页，还可以使用嵌套模板页、嵌入

表达式、服务器控件、模板、数据绑定和定位等。

另外，还支持现有的 ASP.NET 程序，MVC 让开发人员可以使用如窗体认证和 Windows 认证、URL 认证、组管理和规则、输出、数据缓存以及 Session 等。



随着读者对本书学习的深入，将会了解并掌握更多 ASP.NET MVC 2.0 的技术，在这里就不再详述了。

1.3.2　实例描述

对于在某个特定的时间的每种语言、框架、工具和平台，都不可避免地存在争论，ASP.NET WebForm 与 MVC 也是如此。

自从 ASP.NET MVC 的第一个测试版本出现时起，关于它与传统 ASP.NET WebForm 的争论就从来没有停止过。

1.3.3　实例应用

通过论坛、投票或者 Blog 中的评论不难看出，目前关于 ASP.NET MVC 最引人关注的问题是它的发布意味着 WebForm 的终结。

但事实并非如此，ASP.NET MVC 并不是 ASP.NET WebForm 4.0。它是 WebForm 的替代方法，也是.NET 架构完全支持的部分。在 WebForm 继续进行新的改革和发展的同时，ASP.NET MVC 也将作为 Microsoft 完全支持的并行替代方法而继续发展。

甚至在一个项目中可以混合使用这两种技术，图 1-2 为它们的技术架构图。

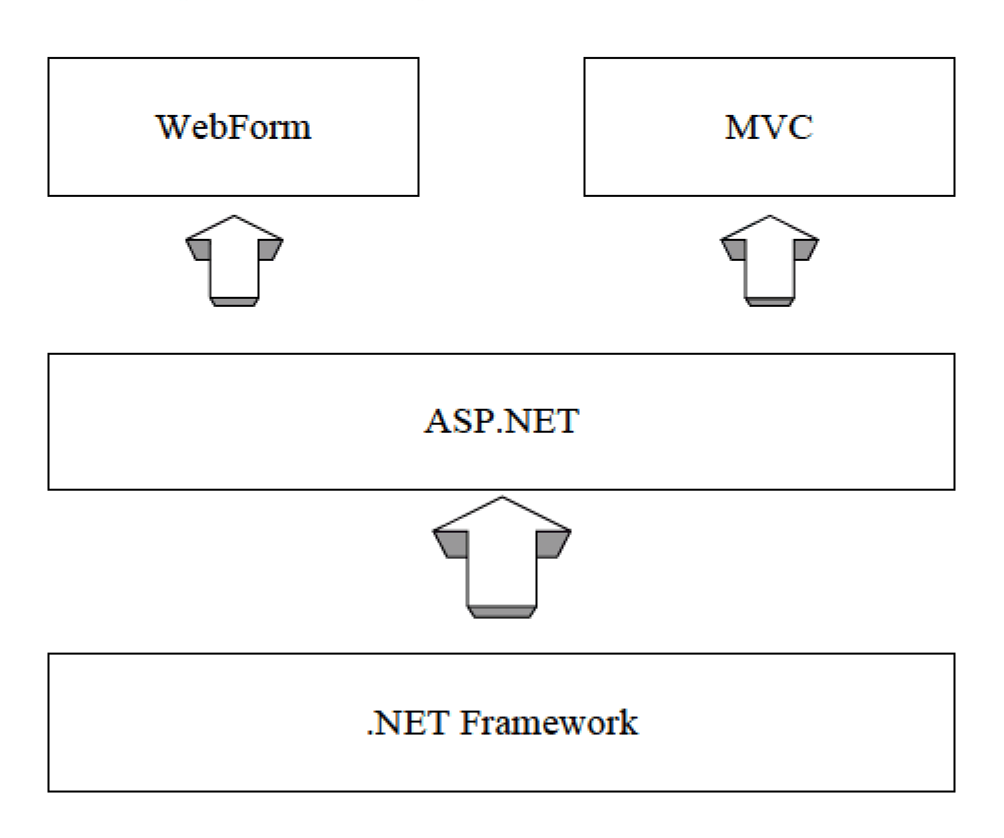


图 1-2　WebForm 与 MVC 技术架构图

如果要验证这一点也非常简单，那就是查看这些技术所处的命名空间。

众所周知，ASP.NET 程序所用的命名空间为 System.Web。ASP.NET MVC 也位于这个命名空间下，完整名称是 System.Web.Mvc，它既不是 System.Mvc，也不是 System.Web2。

1．基于 MVC 的 Web 应用程序的优点

ASP.NET MVC 框架具有以下优点：

- 通过将应用程序分为模型、视图和控制器，化繁为简的工作更加轻松。

- 它不使用视图状态或基于服务器的窗体，使得 MVC 框架特别适合想要完全控制应用程序行为的开发人员。
- 它使用一种通过单一控制器处理 Web 应用程序请求的前端控制器模式，使用户可以设计一个支持丰富路由基础结构的应用程序。
- 它为测试驱动的开发(TDD)提供了更好的支持。
- 它非常适合大型开发人员团队支持的 Web 应用程序，以及需要对应用程序行为进行极度控制的 Web 设计人员。

2. 基于 WebForm 的 Web 应用程序的优点

基于 WebForm 的框架具有以下优点：

- 它支持通过 HTTP 保留状态的事件模型，有益于开发业务线 Web 应用程序。基于 WebForm 的应用程序提供了在数百个服务器控件中受支持的许多事件。
- 它使用页面控制器模式向单个页面添加功能。
- 它针对基于服务器的窗体使用视图状态，使得管理状态信息更加轻松。
- 它非常适合想要利用大量组件快速开发应用程序的 Web 开发人员和设计人员的小型团队。
- 通常，对于应用程序开发而言，它比较简单，这是因为组件(Page 类、控件等)紧密集成并且通常需要比 MVC 模型更少的代码。

3. 对测试驱动的开发的支持

使用 MVC 模式除了可以化繁为简外，还可以使应用程序的测试工作比基于 WebForm 的 ASP.NET Web 应用程序的测试工作更加轻松。

为基于 WebForm 的 ASP.NET 应用程序编写自动化测试可能是一项复杂的工作，因为若要测试单个页面，必须实例化应用程序中的页类、所有子控件以及其他相关类。因为运行页面而实例化的类如此之多，所以可能难以编写专门侧重于应用程序单个部件的测试。

因此，与 MVC 应用程序测试相比，基于 WebForm 的 ASP.NET 应用程序的测试更加难以实现。而且，基于 WebForm 的 ASP.NET 应用程序的测试需要 Web 服务器。MVC 框架可使组件分离并大量使用接口，这样便可以将单个组件与框架的其余部分分开测试。

4. 何时创建 MVC 应用程序

既然 MVC 和 WebForm 是 ASP.NET 之上平行的两个开发平台，那么如果要创建一个 Web 应用程序，就必须仔细考虑是使用 ASP.NET MVC 还是使用 ASP.NET WebForm 来实现。

因为 MVC 只是给开发者提供了 Web 应用程序开发的另一种选择，绝不是替代 WebForm 技术。这两种技术在不同的应用场合，具有不同的优、缺点，开发者需要根据自己的实际情况来选择。

1.3.4　实例分析



源码解析

ASP.NET MVC 出现之前 ASP.NET 编程还是以拖放控件为主，虽然其 aspx/aspx.cs 的配合

方式与控件的易用性大大增强了 ASP.NET 的开发速度，但是大量控件视图维护导致的客户端页面庞大，速度缓慢和难以测试等问题，使开发人员呼唤一种轻量级的开发框架，于是 ASP.NET MVC 出现了。

目前 MVC 的最新版本为 2.0。随 Visual Studio 2010 集成，使 MVC 比 WebForm 具有更多优势。

1.4 创建第一个 MVC 项目

学到这里，相信你对 ASP.NET MVC 有所了解了吧，但不能得意忘形了哦。在前面，我们只是了解一些 ASP.NET MVC 的基本概念。

俗话说，理论联系实践。

接下来，我们就该实践练兵了，借助 Visual Studio 这个大家伙来创建第一个 MVC 项目。通过这个案例，深入了解 ASP.NET MVC。



视频教学：光盘/videos/01/1.4 创建第一个 MVC 项目　　长度：12 分钟

1.4.1 基础知识——MVC 核心命名空间

之所以在这里推荐使用 Visual Studio 2010，主要是因为 ASP.NET MVC 的核心命名空间是由.NET Framework 4.0 提供的，该平台上的最佳开发工具是 Visual Studio 2010，而且对 MVC 2.0 的支持最完整，功能也很全面。

下面我们来看一下 ASP.NET MVC 2.0 都有哪些比较核心的命名空间。具体查看方式是在 Visual Studio 2010 中创建一个 MVC 项目，然后打开项目的【引用】节点，如图 1-3 所示。

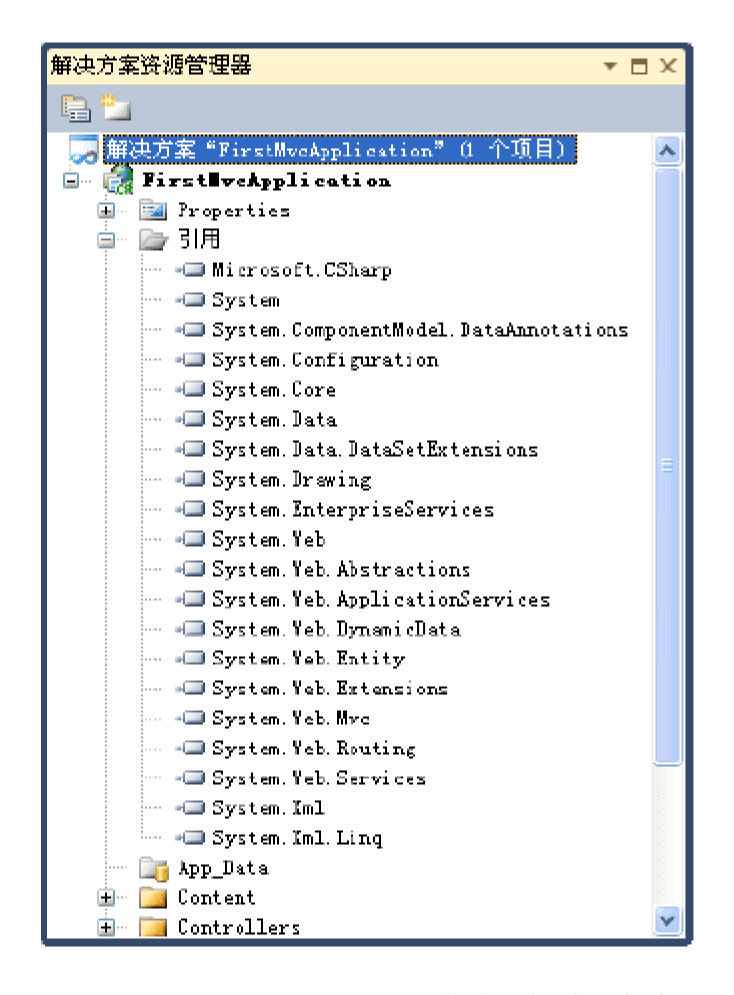


图 1-3 MVC 引用核心命名空间

这些与 MVC 有关的核心命名空间主要有以下几个。

1) System.Web.Routing

URL 路由在该命名空间下提供了使用 URL 路由功能的类，它可以将一个 URL 路由映射对

应到 Controller 上，而不是映射到一个物理文件。

2) System.Web.Extensions

这是 ASP.NET Ajax 的命名空间，在 MVC 中使用 Ajax 功能时需要引用。

3) System.Web.Mvc

这是 ASP.NET MVC 最主要的命名空间。该命名空间包含一些类和接口，它们支持用于创建 Web 应用程序的 ASP.NET MVC 框架。该命名空间包含表示控制器、控制器工厂、操作结果、视图、分部视图以及模型编译等的类。

4) System.Web.Abstractions

该命名空间包含一些相关的基类，例如 HttpContextBase 和 HttpRequestBase 等。

5) System.Web.DynamicData

该命名空间包含为 ASP.NET 动态数据提供核心功能的类。另外，它还提供允许自定义动态数据行为的扩展性功能。



我们也可以创建一个普通的 WebApplication 项目，然后引用这些命名空间，再配置一下 Web.config 文件，从而手动创建一个 MVC 项目。

1.4.2　基础知识——MVC 应用程序目录结构

ASP.NET MVC 框架会将模型、视图和控制器组件分开。默认情况下，每组组件都位于 MVC Web 应用程序项目的单独文件夹中。

在创建新的 MVC Web 应用程序时，Visual Studio 2010 可让用户选择同时创建两个项目。第一个项目是 Web 项目，将在该项目中实现应用程序。第二个项目是单元测试项目，可以在该项目中为第一个项目中的 MVC 组件编写单元测试。

在创建 ASP.NET MVC Web 应用程序项目时，MVC 组件会按项目文件夹自动分开，如图 1-4 所示。

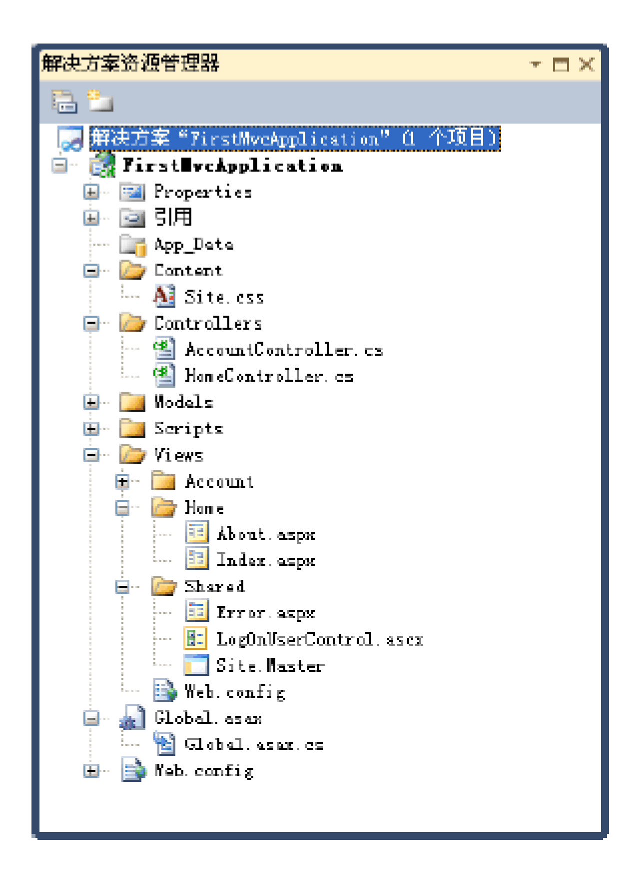


图 1-4　MVC 应用程序目录结构

默认情况下，MVC 应用程序目录结构包括以下文件夹。

- 引用　包含项目运行所需的命名空间与程序集，在上一节中介绍过。

- App_Data　这是数据的物理存储区。此文件夹的作用与它在使用 WebForm 页面的 ASP.NET 网站中的作用相同。
- Content　建议在此位置添加内容文件，如级联样式表文件、图像等。通常，Content 文件夹用于存储静态文件。
- Controllers　建议在此位置存储控制器。MVC 框架要求所有控制器的名称均以 Controller 结尾，如 HomeController、LoginController 或 ProductController。
- Models　这是为表示 MVC Web 应用程序的应用程序模型的类提供的文件夹。此文件夹通常包括定义对象以及定义与数据存储交互所用的逻辑的代码。通常，实际模型对象将位于单独的类库中。但是在创建新应用程序时，可以将类放在此处，然后在开发周期中稍后的某个时刻将其移动到单独的类库中。
- Scripts　建议在此位置存储支持应用程序的脚本文件。默认情况下，此文件夹包含 ASP.NET Ajax 基础文件和 jQuery 库。
- Views　建议在此位置存储视图。视图使用 ViewPage(.aspx)、ViewUserControl(.ascx) 和 ViewMasterPage(.master) 文件，以及与呈现视图相关的任何其他文件。在 Views 文件夹中，每个控制器都具有一个文件夹，该文件夹以控制器名称前缀命名。例如，如果控制器名为 HomeController，则 Views 文件夹包含名为 Home 的子文件夹。

默认情况下，当 ASP.NET MVC 框架加载视图时，它将在“Views\控制器名称”文件夹中寻找具有请求的视图名称的 ViewPage (.aspx)文件。默认情况下，Views 文件夹中也有一个名为 Shared 的子文件夹，但该文件夹不与任何控制器对应。



Shared 文件夹用于存储在多个控制器之间共享的视图。例如，我们可以将 Web 应用程序的母版页放在 Shared 文件夹中。

1.4.3　基础知识——MVC 路由

除了使用前面列出的文件夹之外，ASP.NET MVC Web 应用程序还使用 Global.asax 文件中的代码来设置全局 URL 路由默认值，并且使用 Web.config 文件来配置应用程序。

路由在 Global.asax 文件的 Application_Start 方法中初始化。下面的示例演示了一个包含默认路由逻辑的普通 Global.asax 文件。

```
 public class MvcApplication : System.Web.HttpApplication
 {
     public static void RegisterRoutes(RouteCollection routes)
     {
         routes.IgnoreRoute("{resource}.axd/{*pathInfo}");//   ❶
         routes.MapRoute(//  ❷
             "Default",                          //路由名称
             "{controller}/{action}/{id}",       //带有参数的 URL
            new { controller = "Home", action = "Index", id =
UrlParameter.Optional }                          //参数默认值
         );
     }
     protected void Application Start()
     {
         AreaRegistration.RegisterAllAreas();
         RegisterRoutes(RouteTable.Routes);
```

```
        }
    }
```

在上述代码中定义了两个 URL 路由。在❶处定义了可以忽略的路由配置，也就是说，不需要路由处理程序去处理这些路由，在❷处则设置了一个默认的路由。

当第一次启动 ASP.NET MVC 应用程序时，调用 Application_Start()方法。该方法又调用 RegisterRoutes()和 RegisterRoutes()方法，创建默认的路由表。

默认的路由表中包含一个路由，该默认路由将所有进入的请求拆分为 3 个单元(URL 单元是正斜杠之间的所有内容)。第一个单元映射到控制器名称，第二个单元映射到操作名称，最后一个单元映射到传递给操作名称 ID 的参数。

例如，考虑下面的 URL：

```
/Product/Details/3
```

此 URL 被解析为如下 3 个部分：

```
Controller = ProductController
Action = Details
Id = 3
```



前缀控制器将被附加到控制器参数的末端，这只是 MVC 的一个特殊之处。

默认路由包括所有 3 个单元的默认值。默认控制器是 HomeController，默认操作是 Index，而默认 ID 是一个空字符串。观察这些默认值，考虑如何解析下面的 URL：

```
/Employee
```

此 URL 被解析为如下 3 个参数：

```
Controller = EmployeeController
Action = Index
Id = ""
```

最后，如果打开 ASP.NET MVC 应用程序而不提供任何 URL(例如 http://localhost/)，那么 URL 将被解析为：

```
Controller = HomeController
Action = Index
Id = ""
```

请求将被发送到 HomeController 类的 Index()操作上。

1.4.4　基础知识——MVC 项目中的模型、视图与控制器

在构建传统的 ASP.NET WebForm 应用程序时，URL 和页面是一一对应的。如果从服务器上请求名称为 Index.aspx 的页面，则硬盘上最好有名称为 Index.aspx 的页面。如果 Index.aspx 文件不存在，则将出现 404 － Page Not Found 错误。

相反，在构建 ASP.NET MVC 应用程序时，在浏览器地址栏中键入的 URL 和应用程序中的文件不存在对应关系。在 ASP.NET MVC 应用程序中，URL 对应的是控制器操作，而不是硬盘上的页面。

还有，在传统的 ASP.NET 应用程序中，浏览器请求是映射到页面上的。在 ASP.NET MVC 应用程序中，浏览器请求是映射到控制器操作。ASP.NET WebForm 应用程序关注的是内容，而 ASP.NET MVC 应用程序关注的则是应用程序逻辑。

1. 控制器

控制器负责控制用户与 MVC 应用程序的交互方式。控制器决定在用户发出浏览器请求时向用户发送什么样的响应。

控制器只是一个类。ASP.NET MVC 示例应用程序包括一个名称为 HomeController.cs 的控制器，该控制器位于 Controllers 文件夹内。

HomeController.cs 的内容如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace FirstMvcApplication.Controllers
{
    [HandleError]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Message"] = "欢迎使用 ASP.NET MVC!";
            return View();
        }
        public ActionResult About()
        {
            return View();
        }
    }
}
```

从以上代码可以看到，HomeController 有两个方法，名称为 Index()和 About()。这两个方法对应于控制器公开的两个操作。/Home/Index 调用 HomeController.Index()方法，而 /Home/About 调用 HomeController.About()方法。

控制器中的任何公共方法都作为控制器操作被公开，这意味着通过向浏览器输入正确的 URL 来访问 Internet 的任何人都可以激活包含在控制器中的任何公共方法。

2. 视图

由 HomeController 类中公开的两个控制器方法 Index()和 About()都返回一个视图。视图包括发送到浏览器的 HTML 标记和内容。在使用 ASP.NET MVC 应用程序时，视图等于页面。

因此，必须在正确的位置创建视图。HomeController.Index()操作返回位于以下路径的视图：

```
\Views\Home\Index.aspx
```

HomeController.About()操作返回位于以下路径的视图：

```
\Views\Home\About.aspx
```

总之，如果要为控制器操作返回视图，则需要在 Views 文件夹中使用与控制器相同的名称创建子文件夹。在子文件夹中，必须创建与控制器操作名称相同的.aspx 文件。

如下为 About.aspx 视图文件中包含的代码：

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage" %>
<asp:Content ID="aboutTitle" ContentPlaceHolderID="TitleContent"
runat="server">
    关于我们
</asp:Content>
<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent"
runat="server">
    <h2>关于</h2>
    <p>
        将内容放置在此处.
    </p>
</asp:Content>
```

如果忽略上述代码中的第一行，则其余大多数视图的内容都由标准的 HTML 组成。在此，可以通过输入任何想要的 HTML 来修改视图的内容。



视图非常类似于 ASP.NET WebForm 的页面，视图包括 HTML 内容和脚本，可以使用我们熟悉的.NET 编程语言(如 C#或 Visual Basic .NET)来编写脚本。可以使用脚本显示动态内容，例如数据库中的数据。

3. 模型

前面，我们已经讨论了控制器和视图。下面要讨论的最后一个主题是模型。

什么是 MVC 模型？

MVC 模型包含所有视图或控制器不包含的应用程序逻辑。模型应该包含所有应用程序业务逻辑和数据库访问逻辑。例如，如果正在使用 LINQ to SQL 访问数据库，那么将在 Models 文件夹中创建 LINQ to SQL 类(dbml 文件)。

视图应该只包含与生成用户界面相关的逻辑。控制器应该只包含要求返回正确视图或者将用户重定向到另一操作所需的最小逻辑，其他所有内容都应包含在模型中。

总之，应该努力实现高效模型和简化控制器。控制器方法应该只包含几行代码。如果控制器操作过长，则应该考虑将逻辑移动到 Models 文件夹下的一个新类中。

1.4.5　实例描述

每当我们在学习一门新的编程语言时，都会习惯于编写一个最简单的实例 Hello World 来向这个陌生的世界打一声友好的招呼。

下面我们就来要通过第一个 MVC 实例，快速掌握在 Visual Studio 2010 下创建 MVC 项目的过程以及 MVC 项目的目录结构。

1.4.6 实例应用

【例 1-1】创建第一个 MVC 项目。

在开始创建项目之前，先啰嗦几句。常言道“工欲善其事，必先利其器”，说的就是好的开发工具能够减少我们的很多体力劳动。

开发 ASP.NET Web 应用程序，就需要用到微软开发的一个非常强大的集成开发环境 Visual Studio 2010。

这里之所以使用 Visual Studio 2010，正是因为它是当前最为流行的开发工具，而且安装后集成了 MVC 的 ASP.NET MVC 2.0。

假设大家已经装了 Microsoft Visual Studio 2010 中文版，具体的安装步骤这里不再多说。在强大的安装向导下，相信你一定可以轻松搞定。

下面开始详述具体的创建过程。

(1) 打开 Visual Studio 2010，方法有很多种，选择自己最常用的，步骤略过。

(2) 经过几秒钟华丽的动画加载之后，我们看到了 Visual Studio 2010 IDE 的主界面。

细心的读者可能会觉得，它与以前的 Visual Studio 2008 不怎么像。没关系，其实它一点也不陌生，而且还非常人性化和友好。

(3) 选择【文件】|【新建】|【项目】命令，打开【新建项目】对话框。

(4) 从左侧选择 Web 分类模板，右侧会出现很多 Web 项目列表。我们要创建 MVC，所以选择【ASP.NET MVC 2 Web 应用程序】选项。

(5) 在下方的【名称】文本框中，为要创建的 MVC 项目起一个有意义的名称，这里为 FirstMvcApplication，单击【确定】按钮，如图 1-5 所示。



本实例中选择的“ASP.NET MVC 2 Web 应用程序”模板将会自动创建一个基本型的测试示例。如果选择“ASP.NET MVC 2 空 Web 应用程序”模板，则将仅仅搭建 MVC 环境，并不包含任何其他可执行的代码。

(6) Visual Studio 2010 将会在指定位置创建 MVC 项目，并复制目录和文件，以及进行预定义的配置工作。完成后会弹出图 1-6 所示的【创建单元测试项目】对话框，询问是否创建与 MVC 项目匹配的单元测试项目。

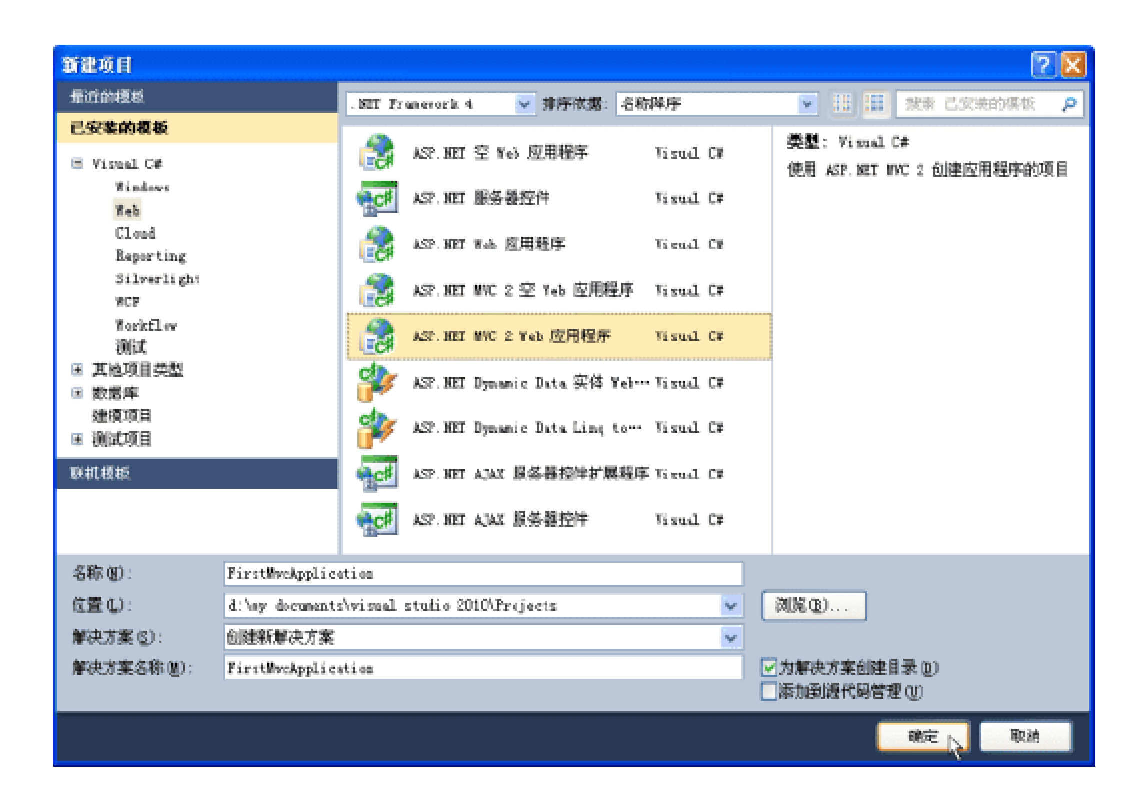


图 1-5 【新建项目】对话框

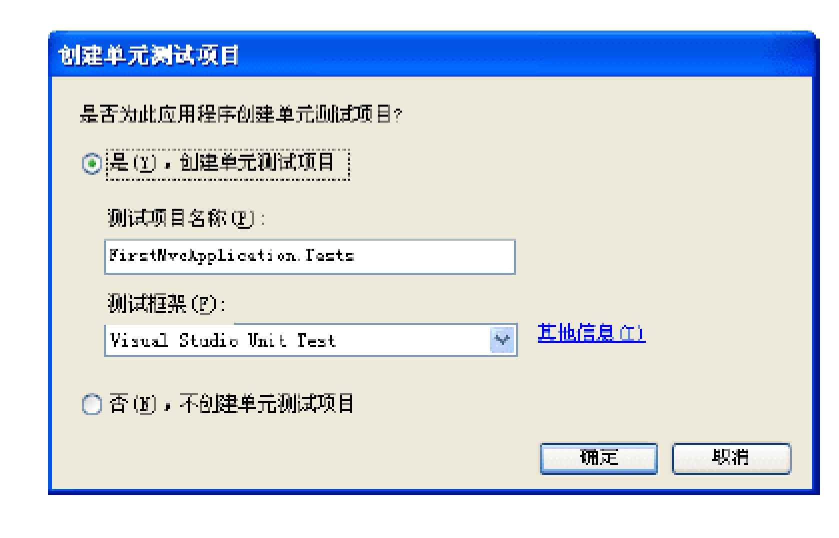


图 1-6 【创建单元测试项目】对话框

由于我们会在下一节重点介绍单元测试的内容，这里选中【否(N)，不创建单元测试项目】单选按钮再单击【确定】按钮。

(7) 最后，单击【确定】按钮完成创建过程。

至此，一个最简单的 MVC 项目就算创建完成了。

1.4.7　运行结果

如果打开【解决方案资源管理器】窗格，会看到 Visual Studio 2010 自动创建的项目结构以及文件内容。这些我们都先不管它，直接运行项目。终于有一个 ASP.NET MVC 项目已经运行在我的电脑上了，是不是很激动啊！

图 1-7 为默认运行后的界面效果，而且“主页”和“关于”功能都能用。图 1-8 为单击“关于”链接后的界面效果。

虽然现在不懂是怎么实现的，不过看着能运行的实例，心里就踏实多了!



图 1-7　默认运行效果



图 1-8　查看“关于”的内容

1.4.8　实例分析

源码解析

如果读者之前使用的是 ASP.NET WebForm，则会发现 ASP.NET MVC 与其非常类似。

通过本实例希望读者能够体会到创建 ASP.NET MVC 应用程序的过程与创建 ASP.NET WebForm 应用程序的相同和不同之处。

本实例介绍了 ASP.NET MVC 应用程序的核心命名空间、目录结构和路由机制。还从高层面讲解了 ASP.NET MVC 中的模型、视图和控制器，以及这些部分如何协同工作。

1.5　创建带单元测试的 MVC 项目

在上一节介绍创建第一个 MVC 项目时，我们认识了 Visual Studio 2010 提供的【创建单元测试项目】对话框。那时我们选择了不创建单元测试项目选项，本节将详细介绍选择创建单元测试项目选项之后所发生的事情。



视频教学：光盘/videos/01/1.5　创建带单元测试的 MVC 项目　　长度：6 分钟

1.5.1　实例描述

ASP.NET MVC 与 WebForm 相比，最大的优势就是当应用程序日益复杂时，可以通过分离模型、视图和控制器来进行单元测试。从而使开发者更容易、及时地发现问题。

在 Visual Studio 2010 中创建一个带单元测试的 MVC 项目是一件非常简单和愉快的事情，下面跟随我来看看具体方法吧。

1.5.2　实例应用

【例 1-2】创建带单元测试的 MVC 项目。

(1) 按照上一节介绍的方法在 Visual Studio 2010 中创建一个名为 MvcApplication1 的 MVC Web 应用程序。

(2) 在弹出的【创建单元测试项目】对话框中选中【是(Y)，创建单元测试项目】单选按钮，【测试项目名称】和【测试框架】均保持默认值，单击【确定】按钮完成创建过程。

(3) 此时，打开【解决方案资源管理器】窗格会发现包含了两个项目。其中，名为 MvcApplication1.Tests 的即为 MVC 的单元测试项目，展开后其目录结构如图 1-9 所示。

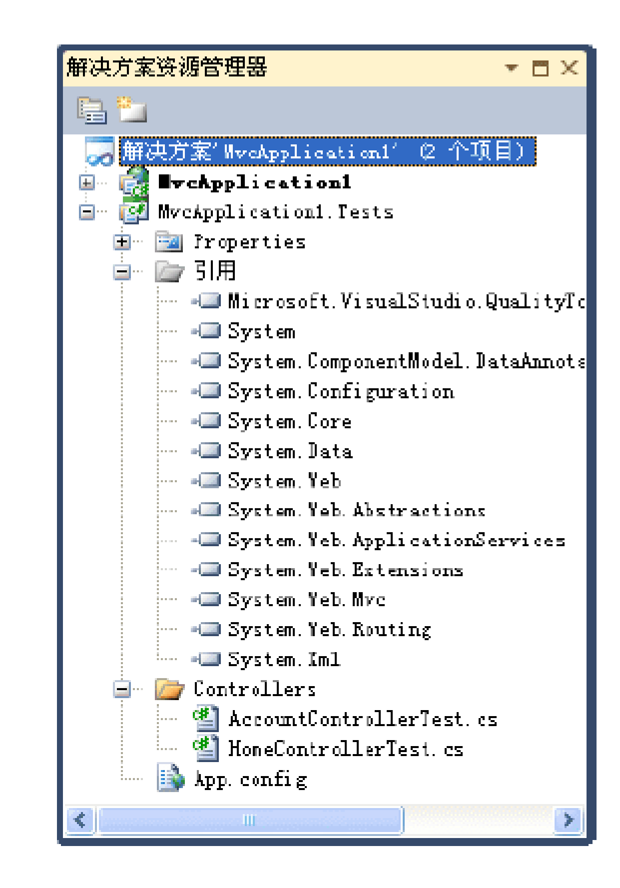


图 1-9　MVC 测试项目目录结构

(4) 可以看到在 Controllers 目录有两个.cs 文件，分别对应 MVC 项目中的控制器。以下为 Home 控制器 Index 动作的测试代码：

```
    [TestClass]
    public class HomeControllerTest
    {
        [TestMethod]
        public void Index()
{
            // 排列
            HomeController controller = new HomeController();
            // 操作
            ViewResult result = controller.Index() as ViewResult;
            // 断言
            ViewDataDictionary viewData = result.ViewData;
            Assert.AreEqual("欢迎使用 ASP.NET MVC!", viewData["Message"]);
        }
    }
```

(5) 在创建普通的 MVC 应用程序时，至此就可以运行了。现在，我们也来运行一下，出错。这是为什么呢？看看有什么提示，图 1-10 为【错误列表】窗格。

(6) 原来是在单元测试项目中找不到对 MVC 项目的引用。

找到原因了。右击单元测试项目选择【添加引用】命令，然后从弹出的【添加引用】对话框中选择项目名称，再单击【确定】按钮，如图 1-11 所示。



图 1-10　【错误列表】窗格

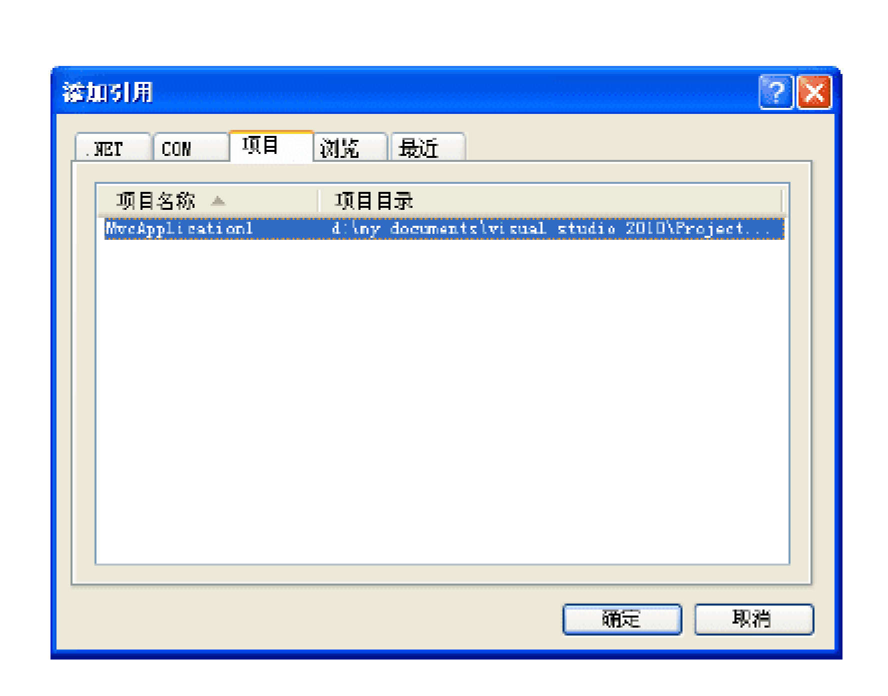


图 1-11　【添加引用】对话框

1.5.3　运行结果

从【测试工具】工具栏中单击【运行解决方案中的所有测试】按钮运行测试，然后从【测试结果】窗格中可以看到测试运行的状态以及结果。

如图 1-12 所示，可以看到现在针对所有控制器的测试方法都成功了。此时再运行解决方案，错误消失了，出现熟悉的运行界面。

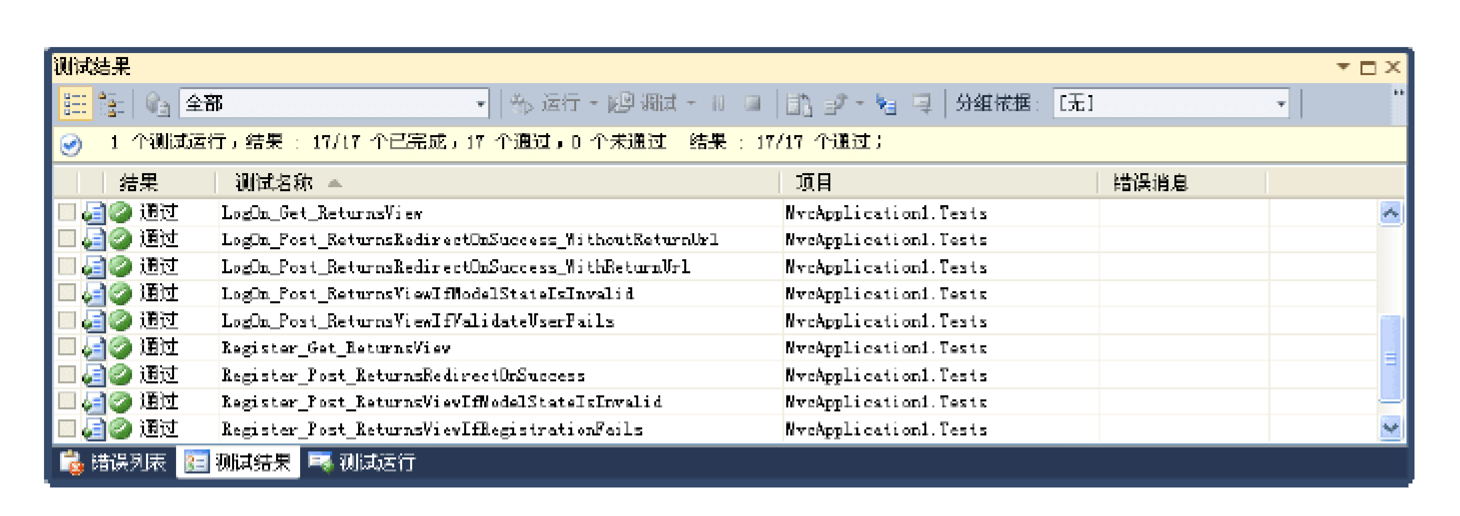


图 1-12　【测试结果】窗格

1.5.4　实例分析



源码解析

在这个实例中，我们学习了如何为一个 MVC 应用程序添加单元测试项目。

单元测试项目的运行依赖于一个 MVC 应用程序，因此必须保证添加了正确的引用。

同时，为了使单元测试项目能够运作正常，在开始执行之前，可以利用测试工具调试解决方案中的所有测试，并在【测试结果】窗格中观察结果。

1.6　ASP.NET MVC 应用程序运行流程

正如前面讲解的 ASP.NET MVC 应用程序的创建过程，在创建之后系统默认创建了相应的文件夹进行不同层次的开发。在 ASP.NET MVC 应用程序的运行过程中，同样请求会被发送到 Controllers 中，这样就对应了 ASP.NET MVC 应用程序中的 Controllers 文件夹；Controllers 只

负责数据的读取和页面逻辑的处理。

在 Controllers 读取数据时，需要通过 Models 从数据库中读取相应的信息。读取数据完毕后，Controllers 再将数据和 Controller 整合并提交到 Views 视图中，整合后的页面将通过浏览器呈现在用户面前。

当用户访问 http://localhost:2236/Home/About 页面时，首先这个请求会被发送到 Controllers 中，Controllers 通过 Global.asax 文件中的路由设置并进行相应的 URL 映射。Global.asax 文件的相应代码如下：

```
     public static void RegisterRoutes(RouteCollection routes)
    //注册路由
     {
         routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
         routes.MapRoute(
             "Default",
             "{controller}/{action}/{id}",
            new { controller = "Home", action = "Index", id =
UrlParameter.Optional }     //配置路由
         );
     }
```

上述代码实现了映射操作，具体是如何实现的，可以先不关心，我们会在下一章讲到它。

我们先来看看 Controllers 和 Views 文件夹中的文件。在 Views 文件夹中包含 Home 子文件夹，在 Home 子文件夹中存在 About.aspx 和 Index.aspx 文件。在 Controllers 文件夹中包含了与 Home 子文件夹同名的 HomeController.cs 文件。

当用户访问 http://localhost:2236/Home/About 时，首先该路径请求会被传送到 Controller 中。

在 Controller 中，Controller 通过 Global.asax 文件和相应的编程来实现路径的映射，示例代码如下：

```
public ActionResult About()              //实现 About 页面
{
    return View();                       //返回视图
}
```



在 Controllers 文件夹中创建 HomeController.cs 文件同 Home 是同名文件，在 Controllers 中创建的文件，其文件名后的 Controller.cs 是不能更改的，所以 HomeController.cs 文件也可以看做是 Home 子文件夹的同名文件。

上述代码实现了 About 页面的页面呈现，在运行相应的方法后会返回一个 View。这里默认返回的是与 Home 的 About 方法同名的页面，即 About.aspx。

将 About.aspx 页面中的文字进行相应的更改，修改后的代码如下：

```
<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent"
runat="server">
    <h2>关于我们</h2>
    <p>
        这是一个 ASP.NET MVC 示例程序的关于我们页面.
    </p>
</asp:Content>
```

运行 About.aspx 页面，结果如图 1-13 所示。



图 1-13　About.aspx 页面

从上述代码可以看出，Controllers 与 Global.asax 用于 URL 的映射，而 Views 用于页面的呈现。从这里还可以看出，当用户访问 http://localhost:2236/Home/About 时，访问的并不是服务器中的/Home/About 页面，而是 Controllers 中的 HomeControllers 的 About 方法。



ASP.NET MVC 应用程序中的 URL 路径访问的并不是一个页面，而是一个方法。例如，访问/Home/About 实际访问的是 HomeControllers 中的 About 方法，而访问/Account/Login 就是访问 AccountControllers 中的 Login 方法。

在这个默认创建的 ASP.NET MVC 示例应用程序中，对应关系如图 1-14 所示。

在 ASP.NET MVC 应用程序中，HomeController.cs 对应 Views 的 Home 子文件夹，而其中的 Index 和 About 方法分别对应于 Index.aspx 文件和 About.aspx 文件。

最后，实现相应的 URL 映射需要通过修改 Global.asax 文件进行实现。具体如何通过修改 Global.asax 文件进行不同的 URL 映射将在下一章讲解。

```
HomeController.cs ———————————————— Home

public ActionResult Index() ———————— Index.aspx
{
    ViewData["Message"] = "欢迎使用 ASP.NET MVC!";
    return View();
}

public ActionResult About() ———————— About.aspx
{
    return View();
}
```

图 1-14　MVC 应用程序对应关系图



在命名时，默认情况下 XXXController.cs 对应 Views 的 XXX 子文件夹，而其中 XXXController.cs 的 YYY()方法对应 XXX 子文件夹中的 YYY.aspx，而访问路径 XXX/YYY 即是访问 XXXController.cs 中的 YYY()方法。

1.7　常见问题解答

1.7.1　ASP.NET MVC 的初级问题



关于 ASP.NET MVC 的初级问题
网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

我想请教一下 ASP.NET MVC 是用来做什么的？

主要还是用于大中型系统吗？

一个.NET 开发的初学者适合接触 ASP.NET MVC 吗？

【解决办法】MVC 的开发效率很高，可以真正做到前后台分离。

MVC 的本质就是将业务逻辑层和业务视图分离，将数据层的数据通过逻辑层传递给视图。

这样对于一般应用没有什么太大的优点，主要就是在企业级开发中，可以更好地做到任务分工，还有就是当系统庞大时更加易于维护。

如果要是初学者建议从 WinForm 学起，因为那里可以学到更多的基础知识。当你有了一定的基础，再来看 MVC 时，就会觉得这个框架很吸引人，尤其是在 Ajax 上。

1.7.2 ASP.NET MVC 的编译软件是什么



ASP.NET MVC 的编译软件是什么？

网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

新手问：ASP.NET MVC 的编译软件是什么？去哪里下载？是 Visual Studio 2008 吗？

【解决办法】还是我这个.NET 的高手来说说吧。

MVC 编译软件？楼主指的应该是开发时用的软件吧。

首先告诉你，.NET 的 MVC 目前已经出了第二个正式版本了，也就是说有 MVC 1.0 和 MVC 2.0。由于这两个版本不向后兼容，所以相当麻烦。很多用 MVC 1.0 开发出来的项目用 MVC 2.0 打不开，但是这两个版本可以装在一起，互不干扰。

在开发工具上，目前 Visual Studio 最高版是 2010 版，它本身集成了 MVC 2.0，推荐使用。选择 Visual Studio 2008 也可以，但是需要先安装 SP1 补丁，再分别安装 MVC 1.0 和 MVC 2.0 的安装包，步骤比较麻烦。

最后，祝楼主一切顺利。选择.NET 就选择了舒适，但也选择了痛苦。

1.8 习　　题

一、填空题

(1) MVC 设计模式将应用程序按用户界面的功能划分为模型、视图和__________3 个模块。

(2) 三层架构里的__________主要是对业务实体数据的加工，把加工后的数据传给页面显示。

(3) ASP.NET MVC 的前身是__________，是受到 Ruby on Rails 的灵感启发。

(4) __________命名空间包含表示控制器、控制器工厂、操作结果、视图、分部视图以及模型编译等的类。

(5) 在创建一个 ASP.NET MVC 应用程序后，__________文件夹用于存储静态文件，__________文件夹存储支持应用程序的脚本文件。

(6) ASP.NET MVC Web 应用程序使用__________文件中的代码来设置全局 URL 路由默认值。

二、选择题

(1) MVC 模式是一个处理__________层逻辑的设计模式。

A. 数据访问　　B. 数据模型
C. 用户界面　　D. 业务逻辑

(2) ASP.NET MVC 是一个__________。

A. 设计思想　　B. 类
C. 框架　　D. 设计模式

(3) 下列说法错误的是__________。

A. ASP.NET MVC 中 View 默认放在 Views 目录下面，也可以是其他目录
B. ASP.NET MVC 中 Model 必须放在 Models 目录下面
C. ASP.NET MVC 中脚本文件必须放在 Scripts 目录下面
D. ASP.NET MVC 中 Controller 默认必须放在 Controllers 目录下面

(4) 默认的 ASP.NET MVC 2 站点访问路径是__________。

A. /Home/Index　　B. Default.aspx
C. Home.aspx　　D. Index

(5) 下列选项中不属于 MVC 缺点的是__________。

A. 由于应用于模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性
B. 视图与控制器间过于紧密的连接
C. 模型返回的数据不带任何显示格式
D. 视图对于模型数据的低效率访问

(6) 下列组合中不属于 MVC 应用的选项有__________。

A. ASP 和 FastMVC　　B. Java 和 Struts
C. PHP 和 Zend　　D. ASP.NET 和 MonoRail

(7) 下面不属于 ASP.NET MVC 框架优点的是__________。

A. 视图与控制器间过于紧密的连接
B. 支持测试驱动的开发
C. 使用页面控制器模式向单个页面添加功能
D. 使用单一控制器处理 Web 应用程序请求

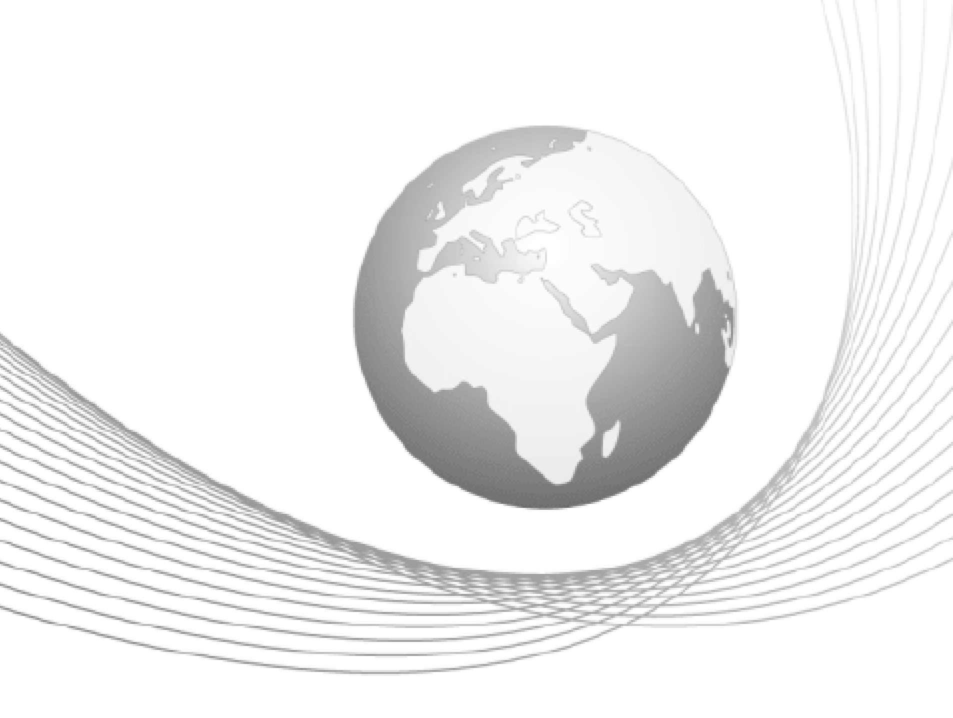
三、上机练习

上机练习： 创建一个带单元测试的 ASP.NET MVC 应用程序。

通过对本章的学习，相信读者一定掌握了 Visual Studio 2010 的基本使用方法。本次练习要求读者在 Visual Studio 2010 的向导下，创建一个 ASP.NET MVC Web 2 应用程序。

要求如下:

(1) MVC 应用程序的名称为 HelloMVC。
(2) 单元测试项目的名称为 TestsHelloMVC。
(3) 测试框架为 Visual Studio Unit Test。



第 2 章　畅通无阻——管理 URLRouting

内容摘要

在 ASP.NET MVC 架构中，抛弃了使用页面文件(如扩展名为.aspx 的页面)接收并处理用户请求的方式，统一使用了继承自 System.Web.Mvc.Controller 类的控制器(Controller)里面的动作(Action)来接收并处理用户的请求。

我们知道，在 ASP.NET MVC 架构中，控制器是一个类，控制器里的动作是一个方法，而 URL 是互联网上标识“资源”的唯一符号，而且这里所说的“互联网上的资源”并不包括 Web 服务器内存中动态存储的一个类和一个方法。如何将 URL 与 Action 对应起来，让用户在执行一个请求的时候正确地执行相应的“动作”来处理用户的请求。ASP.NET MVC 引出了一个新的概念——URLRouting(URL 路由)。

本章我们来学习 URLRouting 的配置规则，了解 URLRouting 和 URL 重写的区别，并学会使用工具调试 URLRouting。

学习目标

- 掌握 URLRouting 的配置规则
- 了解 URLRouting 和 URL 重写的区别
- 学会使用工具对自己配置的 URLRouting 进行调试

2.1　URLRouting 介绍

什么是 URLRouting？直接翻译过来就是 URL 路由。

什么是 URL 路由？简单地说，URL 路由就是一组从 URL 到请求处理程序间的映射规则，用于将 Web 请求引导到实际的请求处理程序中，它在整个 Web 请求过程中担任着向导的作用。

**视频教学：**光盘/videos/02/2.1　什么是 URLRouting　　长度：6 分钟

2.1.1　什么是 URL

要学习 URL 路由，我们先来了解一下什么是 URL。

URL 是 Uniform Resource Locator 的缩写，翻译过来就是“统一资源定位器”。它是用于完整地描述 Internet 上的网页或其他资源的地址的一种标识方法。

一般的 URL 可以由 6 部分组成，一般格式如下(带方括号的为可选项)：

```
protocol :// hostname[:port] [/path] [?parameters][#fragment]
```

URL 各部分的说明如下。

- protocol　协议。可以是 HTTP(超文本传输协议)、FTP(文件传输协议)和 HTTPS(安全的超文本传输协议)等。
- hostname　主机名。指在互联网中存放资源的服务器 DNS 主机名或 IP 地址。
- port　端口号。该选项是一个小于 66 536 的正整数，是各服务器或协议约定的通信端口。
- path　路径。一般用来表示一个 Web 站点中的目录或文件资源的地址。
- parameters　参数列表。可以给使用动态网页技术(如 PHP、ASP、ASP.NET 和 JSP 等)制作的网页传递参数。参数形式为以等号=隔开的键/值对，多个参数之间用 And 符号&连接。
- fragment　信息片断。可以直接定位到页面中的某个锚点标记。

例如访问窗内网论坛的某个页面，如图 2-1 所示。



图 2-1　窗内网论坛

图中地址栏显示的就是这样一个 URL。

```
http://bbs.itzcn.com/index.php?iframe=yes
```

在这里，我们使用 HTTP 协议访问了网络中的一台 Web 服务器(bbs.itzcn.com)的 80 端口(默认端口，可以省略)的 index.php 文件。这是一个动态网页文件，我们向其传递一个参数 iframe，参数的值为 yes。index.php 页面在接收到浏览器的请求以后，根据传递过来的参数进行处理，并向浏览器返回响应结果。

好像有点抽象了。不过我们抛开 URL 的概念来看一下浏览器地址栏里的东西，那是什么？别抽象化地想它……

告诉你吧，那是一个“字符串”。虽然答案有点模糊，但却是一个事实。这个字符串可以被浏览器、DNS、目标主机、Web 服务器程序以及相应的请求处理程序分别解析和处理。

再来看一下 URL 是如何工作的，如图 2-2 所示。

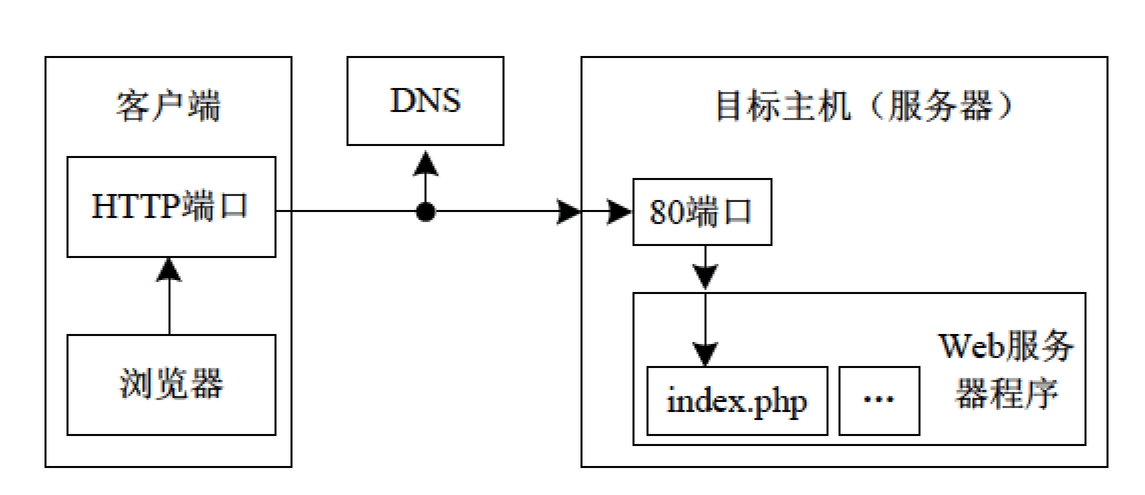


图 2-2　URL 工作原理图

这里必须先明确一点：URL 用来标识一个请求的目标地址。图中箭头所表示的是，当用户在地址栏 URL 中输入 URL 地址并按回车键以后，浏览器请求的走向。

首先浏览器(客户端)截获 URL 中://以前的部分，得知我们要发出一个 HTTP 请求，就使用本机的 HTTP 端口(80)发出这个请求。

中间经过一些网络服务器的转发，来到 DNS。DNS 截获中间的主机地址部分(bbs.itzcn.com)，查询该主机对应的 IP 地址，然后告诉当前请求：你去找某某(这里的“某某”是指主机的 IP 地址)。

当请求来到目标主机对应的 IP 地址的时候，目标主机盛情地接待了它。当目标主机得知这个 HTTP 请求没有明确指出要访问的端口时，就通过默认的端口(80 端口)把它交给相应的 Web 服务器程序(例如 IIS、Apache 等)。

最后，Web 服务器程序把请求交给相应路径的目标处理程序(index.php)。而且，这里的目标处理程序还可以获取 URL 传递过来的参数(iframe=yes)，并进行相应的处理。

通过上面的例子我们看到，这里把 URL 指向服务器上的一个实际的网页文件。但是，到了 ASP.NET MVC 中，URL 就不再是标识一个实际存在的资源路径了，因为我们需要使用 URLRouting。



其实基于 MVC 模式的 Web 应用程序(例如 JSP、PHP 等)也都使用虚拟的 URL。

那么，什么是 URLRouting 呢？它具体是干什么的？下面我们来详细了解一下。

2.1.2　什么是 URLRouting

前面讲过，URLRouting 是一组从 URL 到请求处理程序间的映射规则。

通过前面章节的学习，我们知道在 ASP.NET MVC 架构的应用程序中，真正处理客户端请

求的处理程序是 Controller 中的 Action(使用这种方式的好处在前面的章节中已经详细说明过，这里就不再多讲)，而 Action 是服务器内存中的一个方法。那么，如何将浏览器请求的 URL 指向服务器内存中的这个方法呢？ASP.NET URLRouting 配合 ASP.NET MVC 提供了这个功能。

在上一节中，我们了解了用户可以通过 URL 来访问服务器上的指定资源文件，Web 服务器根据 URL 指定的路径，查找相应的资源文件，这样 URL 和服务器磁盘上的文件有着直接对应的关系。

ASP.NET URLRouting 基本上颠覆了这种 URL 和文件系统一对一的关系，它可以将 URL 直接映射到一个类的方法中调用。

这种接收并处理用户请求的类可以执行业务逻辑，统一调配 Web 系统资源，以及控制应用程序的执行，被称作控制器(Controller)。单单有类，还不能执行操作。要执行操作还需要类里面方法的支持，这些可以处理客户端请求的方法称为动作(Action)。

URLRouting 就是要将 URL 映射到能处理业务需求的 Action 上。

也就是说，在 ASP.NET MVC 应用程序的请求执行流程中，URLRouting 处理的是服务器中请求入口的任务，如图 2-3 所示。

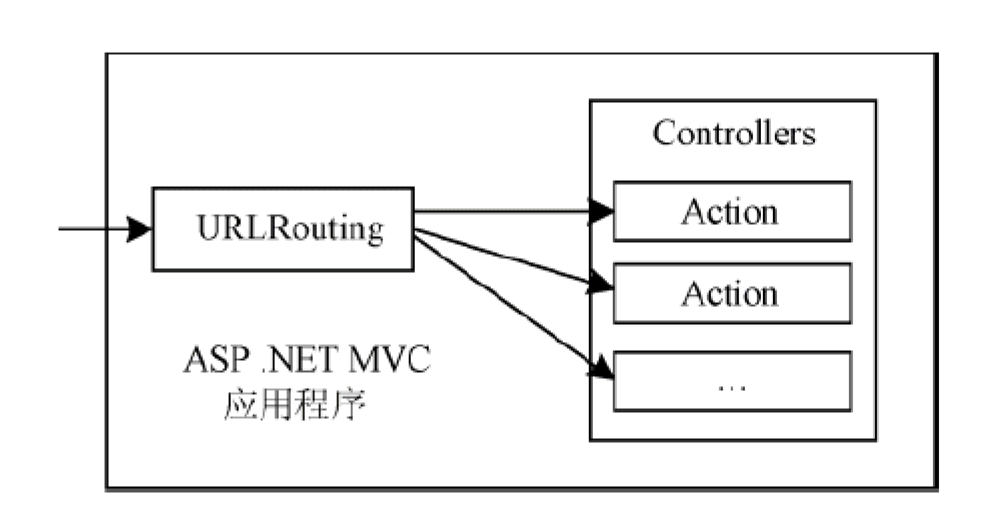


图 2-3 ASP.NET MVC 请求执行流程

在上一节中讲过，URL 只是一个字符串，Web 服务器自动将 URL 映射到相应的服务器资源上。在 ASP.NET MVC 应用程序中，Web 服务器将路由控制权交给 Web 应用程序处理，我们可以编写相应的规则来控制请求的指向。

例如在第 1 章的例子中，在创建应用程序的时候，系统默认为我们配置了一个路由规则，代码如下：

```
routes.MapRoute(
    "Default", // 路由名称
    "{controller}/{action}/{id}", // 带有参数的 URL
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    // 参数默认值
);
```

它可以匹配以“控制器名称”+斜杠+“动作名称”+斜杠+“参数”形式结构的 URL，而且在这个配置里设置了当用户在访问站点首页的时候执行名称为 Home 的控制器里的 Index 动作，所以它还可以匹配没有路径的 URL 访问的 Action(相当于 Web 站点中的 Default.aspx 页面)。



一般来说，在 B/S 项目中，整个 URL 需要 Web 应用程序识别并处理的只有路径和参数部分，所以 URLRouting 的 URL 只需要对路径和参数进行匹配即可。

从上面的配置代码我们看到，URLRouting 将 URL 中的相应部分截取出来进行处理，所以

URLRouting 可以说是一个分析 URL，从 URL 中提取相应数据的组件。

在 ASP.NET 中，实现 URL 路由功能的类被单独封装到一个命名空间下，独立于 ASP.NET MVC 命名空间，如图 2-4 和图 2-5 所示。

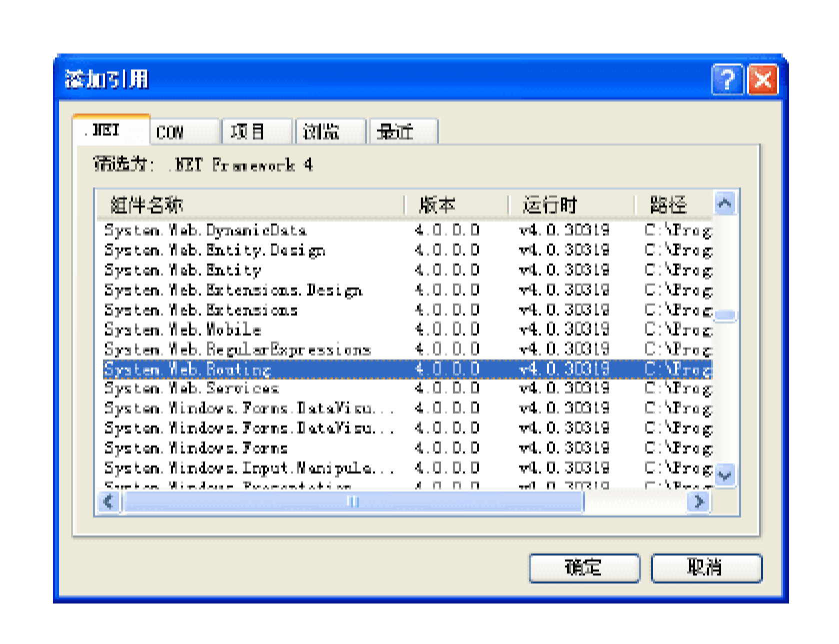


图 2-4　System.Web.Routing 类库引用

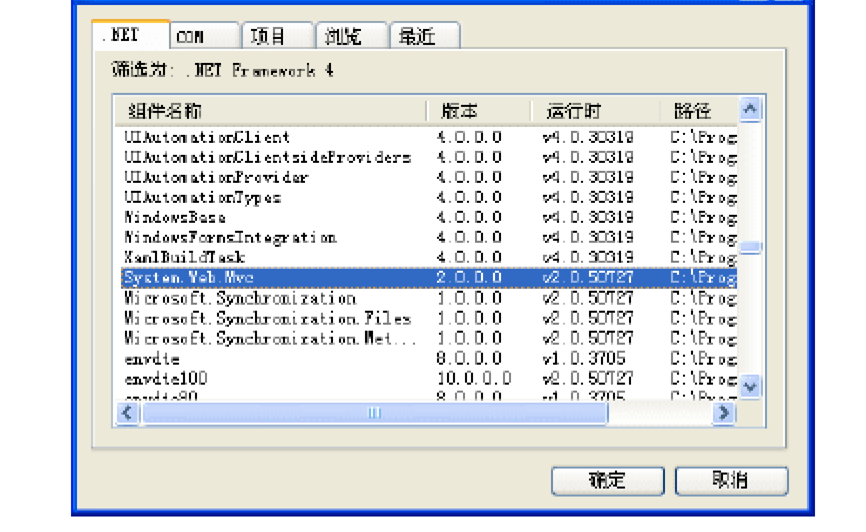


图 2-5　System.Web.Mvc 类库引用

URLRouting 是一个独立的类库，名称为 System.Web.Routing.dll。为了将 URL 映射到 Controller 里的 Action 中，ASP.NET MVC 应用程序需要经常用到它。在 ASP.NET MVC 的类库中，已经集成了对它的引用。



URLRouting 并不是只能用在 MVC 应用程序中，它还可以在 WebForm 应用程序中使用。不过它在 WebForm 应用程序中使用的意义不大，所以这里不做介绍。

另外，ASP.NET MVC 框架是开放源代码的，而 URLRouting 组件目前却不开源，这一点大家可以简单了解一下。

2.2　自定义 URLRouting 规则

通过前面的学习，我们知道每新建一个 MVC 项目，Visual Studio 都会自动在 Global.ascx 文件中定义一个路由规则。使用这个路由规则已经可以匹配大多数的 URL，但是有时候为了业务需求，我们还需要自定义一些满足特定需求的路由规则。

ASP.NET MVC 提供了一套很方便的路由规则定义方案，使得我们可以非常方便地对路由规则进行修改和定义。

 **视频教学：**光盘/videos/02/2.2　自定义 URLRouting 规则　　长度：8 分钟

2.2.1　基础知识

路由(URLRouting 规则)是为了定义如何处理客户端请求。每个 ASP.NET MVC 应用程序至少需要定义一个路由来指明该应用程序如何处理请求。当然，一个非常复杂的 ASP.NET MVC 应用程序很可能包含许许多多的路由。

1. Global.asax 文件

在 ASP.NET MVC 中，默认情况下所有的路由规则都被定义在应用程序根目录下的

Global.asax 文件中。在 ASP.NET 应用程序中，Global.asax 文件是一个应用程序文件，该文件包含响应 ASP.NET 或 HTTP 模块所引发的应用程序级别和会话级别事件的代码，我们经常会使用它来处理一些应用程序(Application)对象的初始化、创建和销毁，以及会话(Session)对象的创建与销毁和请求(Request)对象的创建和销毁之类的事件。

新建一个 ASP.NET MVC 应用程序以后，默认情况下 Global.asax 文件代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcWeb
{
    // 注意：有关启用 IIS6 或 IIS7 经典模式的说明
    // 请访问 http://go.microsoft.com/?LinkId=9394801

    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // 路由名称
                "{controller}/{action}/{id}", // 带有参数的 URL
                new { controller = "Home", action = "Index", id =
UrlParameter.Optional } // 参数默认值
            );
        }

        protected void Application Start()
        {
            AreaRegistration.RegisterAllAreas();

            RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

上面代码为应用程序在应用程序的启动事件(当 HttpApplication 类的第一个实例被创建的时候)添加一个事件处理程序 Application_Start。在该事件处理程序中调用了 AreaRegistration 类的 RegisterAllAreas()方法，注册了 ASP.NET MVC 应用程序的所有区域，然后调用 RegisterRoutes()方法为应用程序注册路由规则。

RegisterRoutes()方法接收一个存储路由规则集合的 RouteCollection 对象，并使用 IgnoreRoute()方法为路由集合添加一个例外的 URL，然后使用 MapRoute()方法为路由集合添加一个路由规则。

2. Route 类

RouteCollection 对象以静态属性的方式声明在 RouteTable 类(命名空间 System.Web.Routing)

中，属性名称为 Routes。

RouteCollection 对象存储的是 Route 类(命名空间 System.Web.Routing)的实例。一个完整的 Route 类实例需要有 URL、默认值、约束、数据代号和路由处理程序等属性。以上各属性声明如下：

```
public RouteValueDictionary Constraints { get; set; }
public RouteValueDictionary DataTokens { get; set; }
public RouteValueDictionary Defaults { get; set; }
public IRouteHandler RouteHandler { get; set; }
public string Url { get; set; }
```

另外，Route 类还有 4 个构造方法，用于初始化路由实例，声明如下：

```
public Route(string url, IRouteHandler routeHandler);
public Route(string url, RouteValueDictionary defaults, IRouteHandler
routeHandler);
public Route(string url, RouteValueDictionary defaults, RouteValueDictionary
constraints, IRouteHandler routeHandler);
public Route(string url, RouteValueDictionary defaults, RouteValueDictionary
constraints, RouteValueDictionary dataTokens, IRouteHandler routeHandler);
```

从上面 4 个构造方法中可知，初始化一个 Route 类的实例首先必须有两个属性 URL 和路由处理程序(routeHandler)，分别用于指定匹配请求的 URL 和处理请求的路由处理程序。

其次，默认值也是使用率非常高的一个属性，所以这里基于第一个构造方法扩充了一个带默认值字典的构造方法。

另外，在实际开发过程中，还需要对路由匹配的参数值进行一些限制，所以这里又扩展了一个带约束字典的构造方法。

最后一个构造方法的参数列表包含了 Route 类的所有属性。但是 DataTokens 属性我们在开发中很少用到，所以最后一个方法不太常用。

3. Route 类的属性

1) URL 属性

在 Route 类中，属性 URL 是一个字符串，用于描述请求中 URL 的格式。该字符串可能不完全是一个实际的 URL，因为其中可以带一些由花括号{}标记的占位符，使用这些占位符可以从 URL 中提取数据(URL 路径或者传递的参数)。例如在 Global.asax 文件中默认添加的 URL 路由规则的 URL 属性如下：

```
"{controller}/{action}/{id}"
```

虽然理论上可以使用任何想要的参数名称来命名路由中的占位符标记，但是为了使 ASP.NET MVC 程序能够正常运转，这里需要约定一些特定的占位符标记，例如{controller}和{action}。

{controller}参数的值用于实例化一个处理请求的控制器类对象。按照约定，该参数的值将被追加一个后缀 Controller，组成一个新的值，并试着寻找名称为该值的 Controller 类，然后实例化该 Controller 类，用来处理请求。注意这里的 Controller 类需要实现 IController 接口(命名空间为 System.Web.Mvc)。

{action}参数的值用于指明处理当前请求将调用控制器里的哪一个方法。这里的方法名只适用于实现了 IController 接口(命名空间为 System.Web.Mvc)的控制器类。



这里的{action}参数所映射的方法必须是 Public(公有的)方法，否则将访问不到。

另外，在参数名称前面添加一个星号即可实现从当前位置起以后的 URL 全匹配。例如以下代码：

```
"{*args}"
```

这句代码将匹配所有形式的 URL。

2) Defaults 属性

Route 类中的 Defaults(默认值)属性是一个以字典的形式存储的“键/值”对集合，它是一个 RouteValueDictionary 类型的对象。

Defaults 属性可以为 URL 属性中的占位符分别指定一个默认值，用于在丢失或者省略一些参数的时候使用。

例如默认配置中的 Defaults 属性代码如下：

```
new { controller = "Home", action = "Index", id = UrlParameter.Optional }
// 参数默认值
```

上面代码使用匿名类的方式为 MapRoute()方法传递一个包含默认值的匿名对象，分别为 URL 的 3 个占位符设置了 3 个默认值。MapRoute()方法在接收到该匿名类的时候，使用反射的方式将其封装成原始格式。

下面我们来了解一下原始的格式。上面说过，Defaults 属性是一个以字典形式存储的“键/值”对的集合 RouteValueDictionary。RouteValueDictionary 类提供了一个构造方法，可以将一个 Dictionary 封装起来，所以这里我们先将数据以“键/值”对的形式保存到 Dictionary 中，再使用 RouteValueDictionary 的构造方法封装成集合对象，代码如下：

```
Dictionary<string, object> defaultDict = new Dictionary<string, object>();
defaultDict["controller"] = "Home";
defaultDict["action"] = "Index";
defaultDict["id"] = 0;
RouteValueDictionary defaultRouteValue = new
RouteValueDictionary(defaultDict);
```

上面代码在数据字典 Dictionary 中封装了 3 个值，然后将该字典封装成一个路由默认值字典。

3) Constraints 属性

Route 类中的 Constraints(约束)属性也是一个以字典的形式存储的“键/值”对集合，它也是一个 RouteValueDictionary 类型的对象。

Constraints 属性可以为对应的 URL 属性中的占位符分别指定一个约束，限制占位符的取值范围。

Constraints 属性的使用方法和 Defaults 属性一致，不过要注意的是 Constraints 属性字典中的值是以正则表达式表示的字符串对象。

技术文档	正则表达式
在计算机科学中，正则表达式是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串。在很多文本编辑器或其他工具里，正则表达式通常被用来检索或替换那些符合某个模式的文本内容。许多程序设计语言都支持利用正则表达式进行字符串操作。 一些常用的通配符说明如下： ● .匹配任何单个字符。例如正则表达式 r.t 匹配的字符串有 rat、rut 和 r t，但是不匹配 root。 ● $匹配行结束符。例如正则表达式 weasel$ 能够匹配字符串"He's a weasel"的末尾，但是不能匹配字符串"They are a bunch of weasels."。 ● ^匹配一行的开始。例如正则表达式^When in 能够匹配字符串"When in the course of human events"的开始，但是不能匹配"What and When in the"。 ● *匹配 0 或多个正好在它之前的那个字符。例如正则表达式.*意味着能够匹配任意数量的任何字符。 ● \这是转义符，用来将这里列出的元字符当作普通的字符来进行匹配。例如正则表达式\$用来匹配美元符号，而不是行尾。 ● |将两个匹配条件进行逻辑或(Or)运算。例如正则表达式(him|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意这个元字符不是所有的程序都支持。 ● +匹配 1 或多个正好在它之前的那个字符。例如正则表达式 9+匹配 9、99 和 999 等。注意这个元字符不是所有的程序都支持。 ● ?匹配 0 或 1 个正好在它之前的那个字符。注意这个元字符不是所有的程序都支持。	

例如我们可以这样对上面的路由规则进行约束：

```
RouteValueDictionary constraintRouteValue = new RouteValueDictionary();
constraintRouteValue["controller"] = @"^\w+";
constraintRouteValue["action"] = @"^\w+";
constraintRouteValue["id"] = @"\d+";
```

这几行代码定义了一个路由约束集合，这里约束该路由的 controller(控制器)必须由一个以上的英文字母组成，action(动作)也必须由一个以上的英文字母组成，参数 id 必须由一位以上的阿拉伯数字组成。



前面使用 RouteValueDictionary 类的构造方法封装了一个 Dictionary 对象，其实 RouteValueDictionary 对象可以像 Dictionary 对象一样使用索引器来访问其内部的数据。

4) RouteHandler 属性

前面说过，URLRouting 组件是独立于 ASP.NET MVC 的，所以其可以配合 ASP.NET MVC 应用程序使用，但在 Route 规则中需要明确指定该路由所指向资源的处理程序。

RouteHandler 属性实现了 IRouteHandler 接口(命名空间为 System.Web.Routing)。

在 ASP.NET MVC 中声明了一个 MvcRouteHandler 类(命名空间为 System.Web.Mvc)，该类实现了 IRouteHandler 接口，所以该类的实例可以作为 RouteHandler 属性的值传入。

4．Route 的使用

前面讲过 Route 类的各个属性及其用法，这里我们使用原始的格式来定义一个路由规则。

以默认的也是最经典的{controller}/{action}/{id}为例，定义代码如下：

```
RouteValueDictionary defaultRouteValue = new RouteValueDictionary();
defaultRouteValue["controller"] = "Home";
defaultRouteValue["action"] = "Index";
defaultRouteValue["id"] = 0;

RouteValueDictionary constraintRouteValue = new RouteValueDictionary();
constraintRouteValue["controller"] = @"^\w+";
constraintRouteValue["action"] = @"^\w+";
constraintRouteValue["id"] = @"\d+";

Route route = new Route(
    "{controller}/{action}/{id}",
    defaultRouteValue,
    constraintRouteValue,
    new MvcRouteHandler());
routes.Add(route);
```

上面代码定义了一个路由规则，配置其 URL 为{controller}/{action}/{id}，默认值参数中设置 controller 的值为 Home，action 的值为 Index，id 的值为 0，并为其添加约束，controller 和 action 都必须是以一个或多个英文字母开头的字符串，参数 id 必须是数字，最后设置路由的处理程序为一个新的 MvcRouteHandler 对象。

好了，我们已经为应用程序配置了一个路由规则。不过看到这里，是不是已经崩溃了——这么啰嗦。

当然，有问题就有解决的方法，也就是系统封装的 RouteCollection 类的 MapRoute()方法。新建项目时系统自动创建的那个 URLRouting 规则就使用了这种简便的方法，初始代码如下：

```
routes.MapRoute(
    "Default", // 路由名称
    "{controller}/{action}/{id}", // 带有参数的 URL
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    // 参数默认值
);
```

MapRoute()方法共有 6 个重载形式，声明分别如下：

```
public static Route MapRoute(this RouteCollection routes, string name, string
url);
public static Route MapRoute(this RouteCollection routes, string name, string
url, object defaults);
public static Route MapRoute(this RouteCollection routes, string name, string
url, string[] namespaces);
public static Route MapRoute(this RouteCollection routes, string name, string
url, object defaults,
                       object constraints);
public static Route MapRoute(this RouteCollection routes, string name, string
url, object defaults,
                       string[] namespaces);
public static Route MapRoute(this RouteCollection routes, string name, string
url, object defaults,
                       object constraints, string[] namespaces);
```

这些方法是一个扩展类 RouteCollectionExtensions(命名空间为 System.Web.Mvc)里的方法，

所以第一个参数 this RouteCollection routes 不必考虑，剩下的就是名称、URL、默认值、约束和数据标记等。



这里的参数“名称”是指为路由规则起的名字，这个名字对于路由的执行无实际意义，只用于开发人员在配置表时更直观地操作路由规则。

使用 MapRoute()方法简化上面的配置代码，结果如下：

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = 0 },
    new { controller = @"^\w+", action = @"^\w+", id = @"\d+" }
);
```

这里同样为应用程序添加了一个路由规则，这个路由规则和上面的配置一样，但是代码量缩减到原来的 1/3，而且更加清晰易读。



因为 MapRoute()方法是 ASP.NET MVC 专门为 RouteCollection 类扩展的方法，该方法只为 ASP.NET MVC 服务，所以这里可以省略掉对 RouteHandler 属性初始化的项。

2.2.2 实例描述

我自己做了一个博客系统，使用的是 ASP.NET MVC 实现的视图架构，也算是对学习 ASP.NET MVC 的总结。

当时在学习之初，对路由规则做了一些配置测试，有一些体会。下面就拿我配置的路由给大家看一下。

2.2.3 实例应用

【例 2-1】自定义 URLRouting 规则。

首先，先预计我的博客需要有以下几种 URL 格式。

- Blog.mvc/2010-11-07　使用该 URL 可以在博客中查询系统中某个日期的所有动态信息。例如博客文章、图片等。
- Photo/life.private　使用该 URL 在相册中相询关键字为 life，访问权限为 private(私有)的照片。
- Photo/2010/work　使用该 URL 查询某年内关于 work 的信息。
- Article/Show/aspnet　使用该 URL 可以访问文章列表，其中关键字为 aspnet。
- Home/Index/32　使用该 URL 访问指定关键字的内容。这里的关键字为整型数值。

了解过了需求，下面着手编写路由代码。为了节省篇幅，这里直接贴出所有配置代码，如下所示：

```
// Blog.mvc/2010-11-07
routes.MapRoute(
```

```
    "BlogRoute",
    "Blog.mvc/{date}",
    new { controller = "Blog", action = "List" },
    new { date = @"^\d{4}-\d{2}-\d{2}" });

// Photo/life.private
routes.MapRoute(
    "PhotoRoute",
    "Photo/{make}.{model}",
    new { controller = "Picture", action = "Show" },
    new { model = @"(private|public)" });

// Photo/2010/work
routes.MapRoute(
    "PhotoRoute2",
    "Photo/{*values}",
    new { controller = "Photo", action = "Index" },
    null);

// Article/Show/aspnet
routes.MapRoute(
    "ArticleRoute",
    "Article/Show/{key}",
    new { controller = "Article", action = "List" },
    new { httpMethod = "POST" });

// Article/Show/aspnet
routes.MapRoute(
    "ArticleRoute2",
    "Article/Show/{key}",
    new { controller = "Article", action = "List" },
    new { });

// Home/Index/32
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = "0" },
    new { controller = @"^\w+", action = @"^\w+", id = @"^\d+" });
```

非常简单就完成了配置。

本节就先演示到这里，至于这些路由如何进行匹配，下一节我们将使用调试工具进行测试。

2.2.4 实例分析



源码解析

本实例根据各种不同的 URL 需求定义了许多不同的路由规则，这里需要注意的是路由规则中的名称参数不可以重复。

上面所定义的路由规则基本上很完美地实现了前面所说的 URL 需求，并且对各种不同的需求添加了相应的默认值和参数约束，更加完善了路由规则的相关配置。

2.3　使用 RouteDebugger 调试路由

上一节我们在 ASP.NET MVC 项目中定义了一系列的 URLRouting。

但是至今为止我们只是从理论上实现了 URL 需求中的各种规则，并没有进行过实际测试。有句话叫“事实胜于雄辩”，在实践中总是会出现一些这样或者那样的错误，所以并不能保证我们的路由规则绝对能够很健壮地执行而不出一点问题。我们不得不想办法对路由规则进行测试。

RouteDebugger 类提供了在 ASP.NET 平台下对路则规则进行调试的功能，本节我们将使用该类对上面所写的 URL 路由进行调试。

**视频教学：**光盘/videos/02/2.3　RouteDebugger 调试路由实验　　长度：10 分钟

2.3.1　基础知识

RouteDebugger 类位于一个单独的类库中，该类库的名称是 RouteDebug.dll。这个类库的体积非常小(才 11KB)，在网上搜索一下就可以找到。

要使用 RouteDebugger 类，需要在项目中添加对该类库的引用，然后导入该类所在的命名空间 RouteDebug。

RouteDebugger 类是一个静态类，而且该类只有一个公有的静态方法 RewriteRoutesForTesting()。

RewriteRoutesForTesting()方法接收一个 RouteCollection 类型的对象作为参数，该参数是一个配置好的路由规则集合，使用该方法可以对该集合内的路由规则进行可视化测试。

2.3.2　实例应用

【例 2-2】使用 RouteDebugger 调试路由。

(1) 首先运行上一节的 MVC 项目。

(2) 我们要对配置过的路由规则进行测试，需要使用到类库 RouteDebug.dll，这一步我们在项目中添加对该类库的引用。

(3) 打开 Global.asax 文件，在页面顶部添加对相应命名空间的引用，代码如下：

```
using RouteDebug;
```

(4) 万事俱备，什么风都不欠了。我们可以直接在应用程序的启动事件中添加对路由规则的测试。这里需要修改 Global.asax 文件中的 Application_Start()方法，修改后的结果如下：

```
protected void Application Start()
```

```
{
    AreaRegistration.RegisterAllAreas();

    /* 初始化路由规则 */
    RegisterRoutes(RouteTable.Routes);

    /* 调试路由 */
    RouteDebugger.RewriteRoutesForTesting(RouteTable.Routes);
}
```

(5) 运行项目，结果如图 2-6 所示。

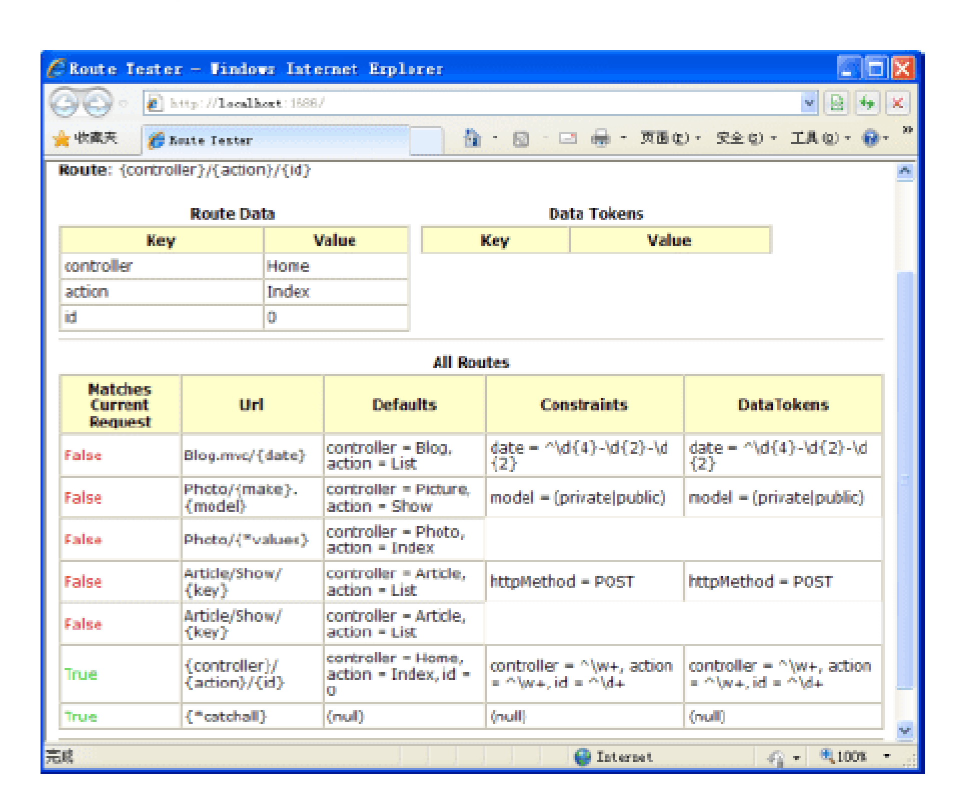


图 2-6　路由调试页面(1)

从运行后的页面我们看到，RouteDebugger 对 URL 请求进行截获，分别在页面中显示出 Route Data、Data Tokens 和 All Routes 等信息。

其中在 All Routes 部分，列出了当前请求的路由测试结果以及路由的 URL、Defaults、Constraints 和 DataTokens 等属性。

在该操作的运行结果中访问了该站点的默认路径，所以该操作匹配了后面两个操作。因为后两个操作中的 URL 路径使用的都是占位符，默认情况下占位符是可以省略的。



在这里我们并没有声明最后一个路由规则，这是 RouteDebugger 自动添加的一个全匹配的路由。

(6) 访问路径 Blog.mvc/2010-11-07 来看一下调试结果，如图 2-7 所示。

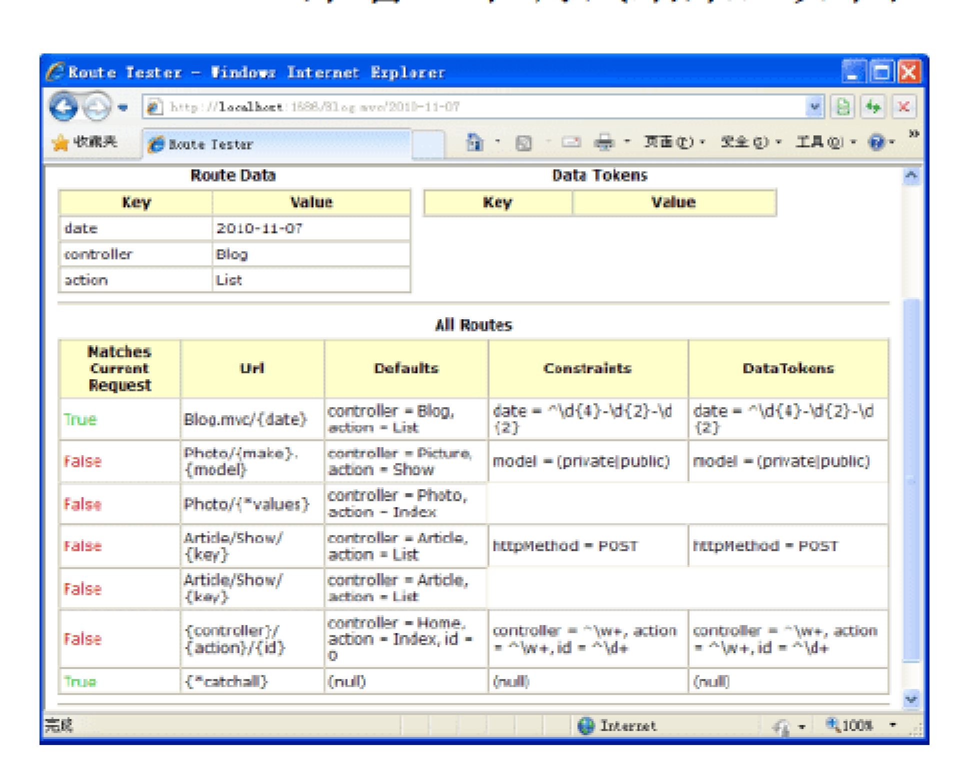
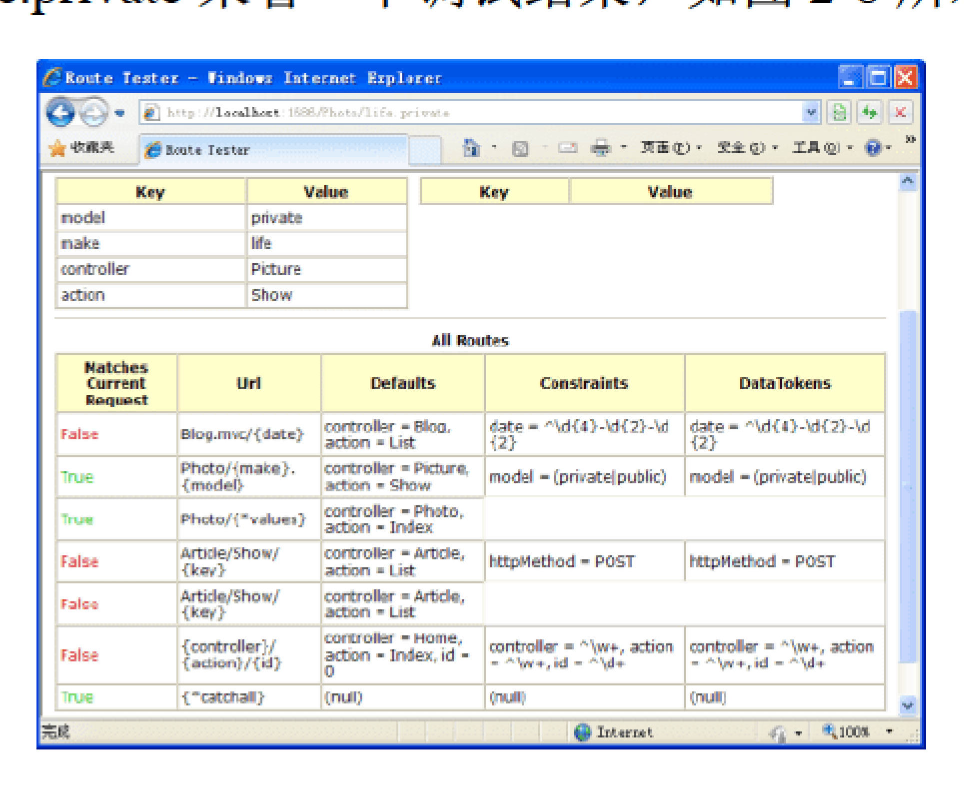


图 2-7　路由调试页面(2)

从执行结果我们看到，该 URL 仅仅匹配了路由规则 Blog.mvc/{date}，与我们预计的一样。

(7) 访问路径 Photo/life.private 来看一下调试结果，如图 2-8 所示。



Key	Value
model	private
make	life
controller	Picture
action	Show

Key	Value

All Routes

Matches Current Request	Url	Defaults	Constraints	DataTokens
False	Blog.mvc/{date}	controller = Blog, action = List	date = ^\d{4}-\d{2}-\d{2}	date = ^\d{4}-\d{2}-\d{2}
True	Photo/{make}.{model}	controller = Picture, action = Show	model = (private|public)	model = (private|public)
True	Photo/{*values}	controller = Photo, action = Index		
False	Article/Show/{key}	controller = Article, action = List	httpMethod = POST	httpMethod = POST
False	Article/Show/{key}	controller = Article, action = List		
False	{controller}/{action}/{id}	controller = Home, action = Index, id = 0	controller = ^\w+, action = ^\w+, id = ^\d+	controller = ^\w+, action = ^\w+, id = ^\d+
True	{*catchall}	(null)	(null)	(null)

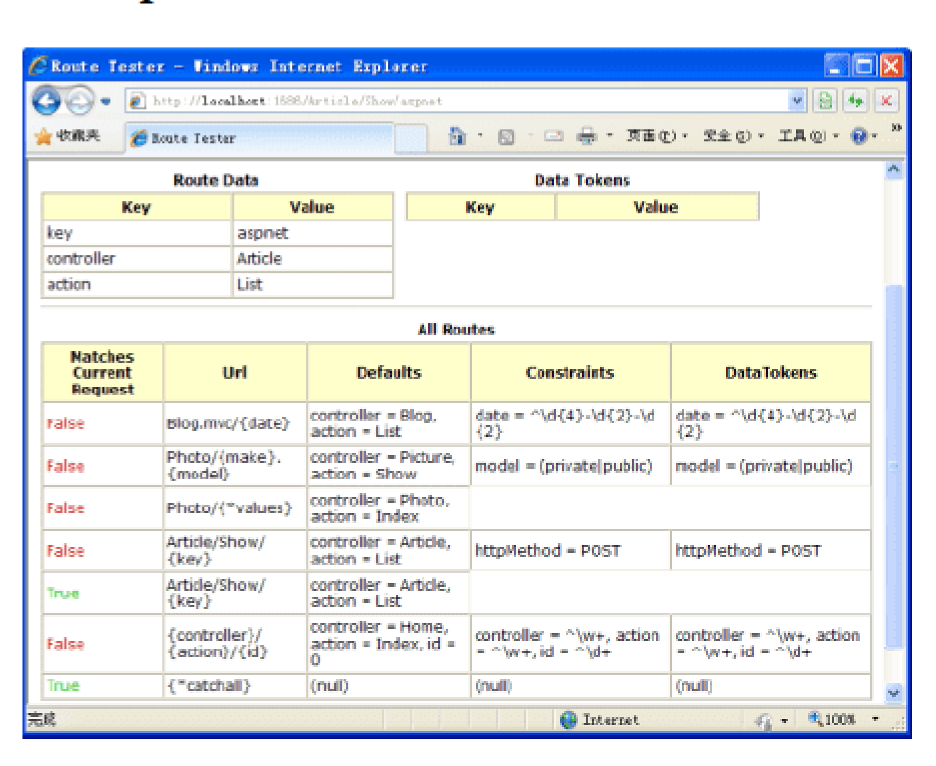
图 2-8　路由调试页面(3)

从执行结果我们看到，该 URL 匹配了 Photo/{make}.{model}和 Photo/{*values}。这是因为该 URL 的结构首先满足了 Photo/{make}.{model}规则，然后 Photo/{*values}使用从当前位置起所有字符全匹配，所以同时能满足这两种 Route。



注意　在所有 URLrouting 中，如果当前 URL 满足多条 Route 配置，则系统默认执行第一个匹配的配置信息。

(8) 访问路径 Article/Show/aspnet 来看一下调试结果，如图 2-9 所示。



Route Data

Key	Value
key	aspnet
controller	Article
action	List

Data Tokens

Key	Value

All Routes

Matches Current Request	Url	Defaults	Constraints	DataTokens
False	Blog.mvc/{date}	controller = Blog, action = List	date = ^\d{4}-\d{2}-\d{2}	date = ^\d{4}-\d{2}-\d{2}
False	Photo/{make}.{model}	controller = Picture, action = Show	model = (private|public)	model = (private|public)
False	Photo/{*values}	controller = Photo, action = Index		
False	Article/Show/{key}	controller = Article, action = List	httpMethod = POST	httpMethod = POST
True	Article/Show/{key}	controller = Article, action = List		
False	{controller}/{action}/{id}	controller = Home, action = Index, id = 0	controller = ^\w+, action = ^\w+, id = ^\d+	controller = ^\w+, action = ^\w+, id = ^\d+
True	{*catchall}	(null)	(null)	(null)

图 2-9　路由调试页面(4)

从执行结果我们看到，该 URL 本来匹配了第四个和第五个 Route，因为它们的 URL 规则都是 Article/Show/{key}。但是，因为第四个 Route 比第五个 Route 多添加了一个特殊的关键字约束 httpMethod，所以我们以普通的 GET 请求方式不能访问该 URL。

(9) 访问路径 Home/Index/32 来看一下调试结果，如图 2-10 所示。

从执行结果我们看到，该 URL 匹配了{controller}/{action}/{id}规则。但是如果说这里的 controller 参数或 action 参数不是英文字母，或者 id 参数不是数字，程序将不会匹配该路由。这里就不再演示了。

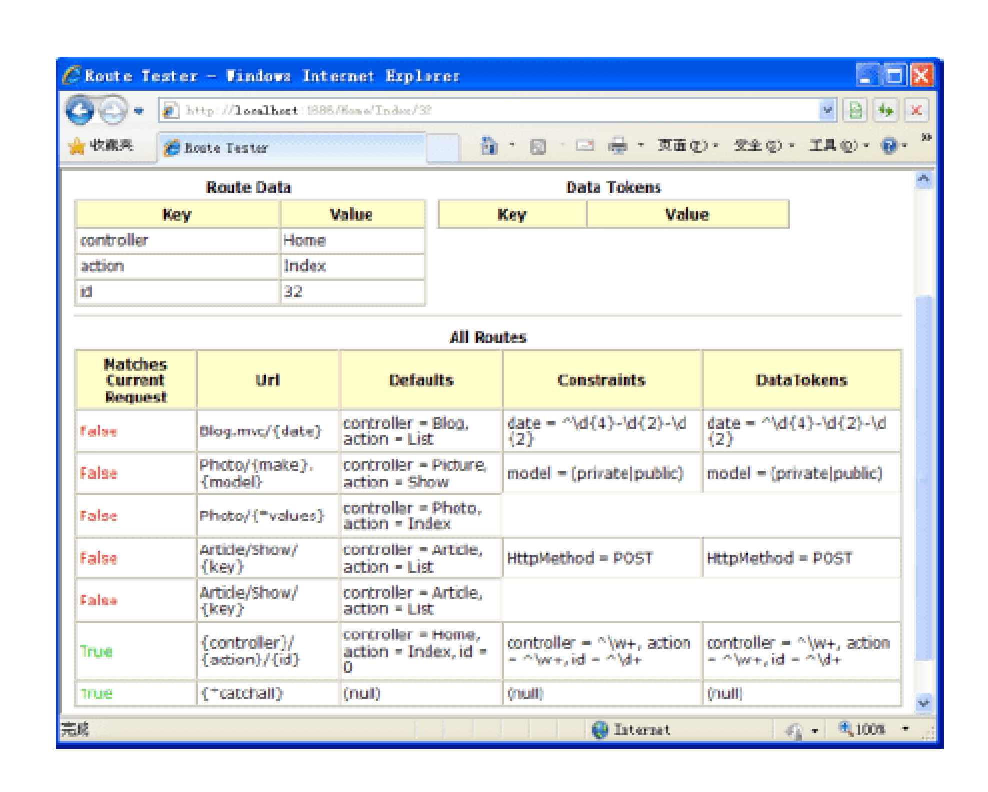


图 2-10　路由调试页面(5)

2.4　URLRouting 和 URLRewrite 的区别

视频教学：光盘/videos/02/ 2.4　URLRouting 和 URLRewrite 的区别　　长度：10 分钟

在学习和使用过 URLRouting 和 URLRewrite 以后，为了更好地了解和区分它们，很多开发人员喜欢将其二者做比较。毕竟两种方法都是在运行时对请求的 URL 进行处理。两者在创建 URL 和代码之间的分离非常有用，都可以帮助我们创建用于搜索引擎优化(Search Engine Optimization，SEO)功能的简洁 URL。



我们可以使用 URLRouting 或 URLRewrite 来组织 URL，使其更加结构化、易读和有利于搜索引擎解析。

关于二者的区别，Ruslan Yakushev 在 LearnIIS.NET 上发表了一篇很有指导意义的文章。文章中说道：二者的本质区别在于 IIS 对 URL 重写的处理方式比 ASP.NET 路由的层次更低，而且对客户端是不可见的。可以看到 URL 重写模块是在请求时被传递到请求处理器(Handler)，例如 ASP.NET 管理的 ASPX 在处理器之前被激活。IIS 的 URL 重写并不知道具体的请求处理器。

此文章同时还给出了 ASP.NET URLRouting 过程的可视化工作流。可以看到 ASP.NET URLRouting 就是一个请求分发器，它必须明确地获知一个特定的请求究竟应途经哪个处理器。

光看文字，可能有点晦涩难懂。我们可以通过它们的用途来了解它们的区别。

首先，前面讲过 URL 是一个标识资源地址的字符串，到服务器中，如何根据 URL 字符串为用户请求执行相应的处理，需要一个中间的映射机制来对系统资源进行统一的调配，如图 2-11 所示。

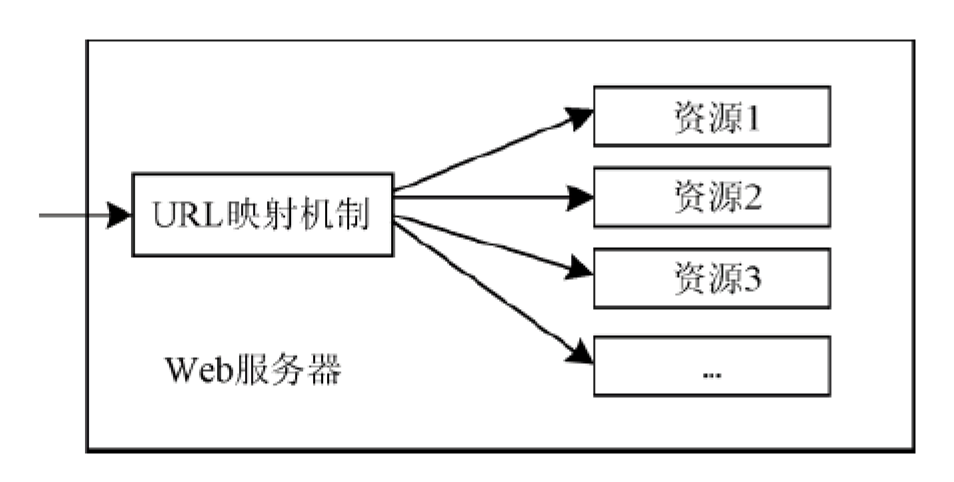


图 2-11　URL 资源映射机制

URL 映射机制(我们先这么叫着)将解析用户请求的 URL，然后根据解析结果调用相应的资源以响应用户的请求。

下面先来看一下 URLRouting。我接触 URLRouting 是在学习 ASP.NET MVC 的时候，姑且就以它在 ASP.NET MVC 中的表现做讲解。

前面我们说过在 ASP.NET MVC 中，URLRouting 是作为服务器中请求的入口，所有的请求都将由它集中处理，转发到相应的资源(这里的资源可能是文件或者 Controller 对象的 Action)。URLRouting 在 ASP.NET MVC 应用程序中的角色如图 2-12 所示。

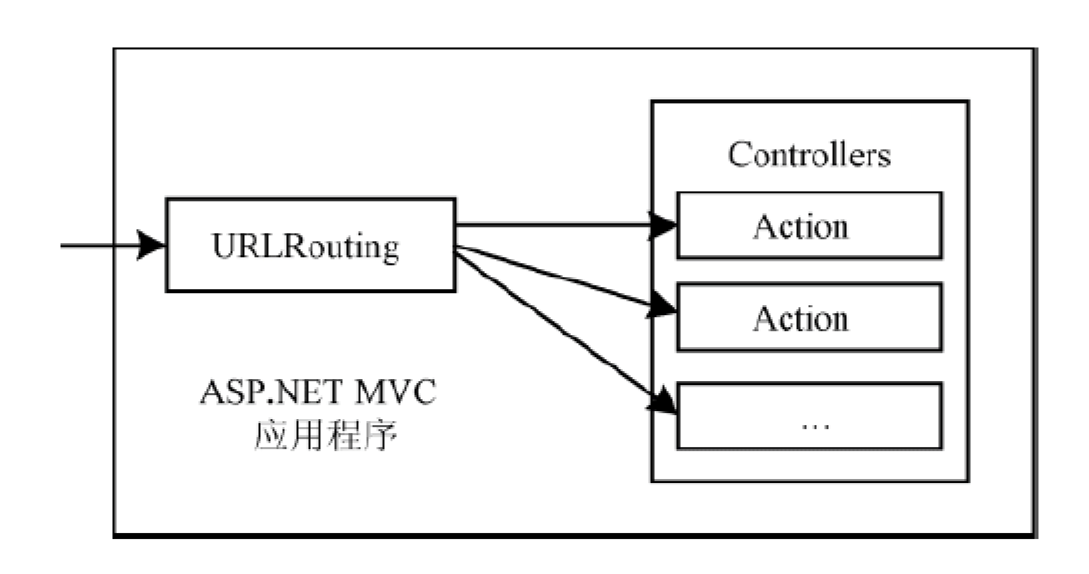


图 2-12　ASP.NET MVC 请求执行流程

可以看到 URLRouting 起着和 URL 映射机制类似的资源映射的功能，而 URLRewrite 则不然。URLRewrite 一般用于改写请求的 URL。例如下面的 URL 地址：

```
/blog/list-3.aspx
```

使用 URLRewrite 可以将其重写成：

```
/blog/list.aspx?cid=3
```

其实这两个 URL 都可以被访问，只是第一个的结构更简单而已。

在 Web 应用中，URLRewrite 的执行流程如图 2-13 所示。

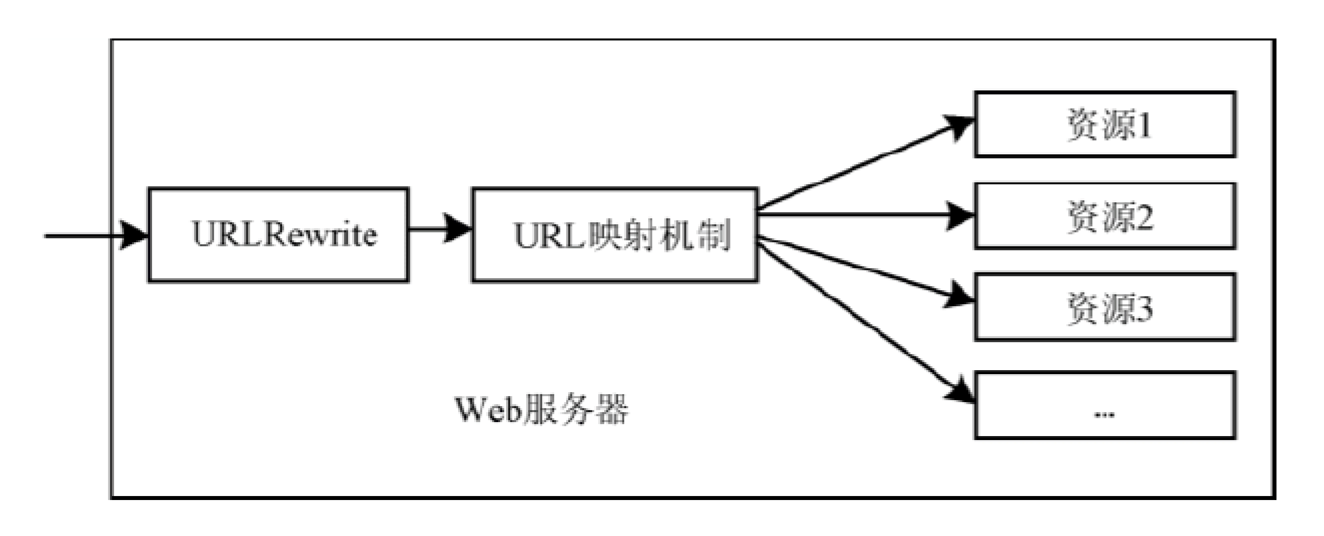


图 2-13　URLRewrite 处理流程图

例如在一个公司里，张总(老板)很忙，所以他交代公司前台：今天有人来找我就让李四经理处理一下。这就相当于 URLRewrite 中定义了一个重写规则：客户访问张总，交给李四经理处理一下。

这个时候，如果有人(不熟悉的人)来找张总处理问题，公司的前台就去请李四经理出来帮用户处理问题。问题处理完了，用户走了，可是用户要找的张总却是一直没有露面。也可能张总不在公司，照样却有人代张总将问题给处理了。

2.5 常见问题解答

2.5.1 能否把 URLRouting 的配置信息保存到 XML 文件中



我能不能把 URLRouting 的配置信息保存到 XML 文件中呢？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

平常我们使用一些框架配置信息的时候通常是将配置信息保存到 XML 文件中，ASP.NET MVC 的 URLRouting 是将路由配置代码写入 Global.asax 文件中。

问题 1：这样用总是感觉不好，我能不能把它单独放在一个 XML 文件中呢？

问题 2：如果可以，我应该怎么做？

【解决办法】

问题 1：当然可以。

问题 2：路由信息其实是写了一些配置信息在代码中，其他一些框架总是会将配置信息写到 XML 中，其实现思路也是一样的，只是微软不建议我们这么做。

首先，你的应用程序中的 URLRouting 配置信息多长时间改一次？经常改吗？应该不是吧。所以 URLRouting 配置一般都是一劳永逸的工作。

但是如果你真的想配置在 XML 文件中，我可以给你一个简单的思路。

(1) 首先需要定义你的 XML 文件结构，以便你以后容易使用代码进行分析。

(2) 使用 XML 反序列化或者直接读取 XML 文件，取出配置信息。这一点你要了解序列化的知识或者 XML 的操作技术。

(3) 使用反射技术将读取出来的配置信息封装到匿名对象中。

(4) 最后添加到路由集合中。

这是大概的思路，你可以到网上查阅相关的资料。网上好像也有现成的例子可供参考，百度一下你就知道了。

2.5.2 具体系统的 URLRouting 配置会不会很多



一个具体的系统，URLRouting 配置会不会很多？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

在一个具体的应用程序中，URLRouting 是不是非常多？

在 Java 中，使用 SSH 框架做的项目往往几百上千行甚至更多的配置信息，ASP.NET 是不是也这样呀？

这时把配置信息放在 Global.asax 文件中会不会很乱？

【解决办法】关于 URLRouting 会不会很多的问题，那就要看你的系统的复杂度了，如果说你的系统非常简单，就没必要配置太多的路由信息，甚至使用默认的就完全可以了。

另外你要是觉得配置信息放在 Global.asax 文件中会很乱，完全可以把它放到其他文件中。例如建立一个实例类，提供一些相应的方法，然后在 Application_Start 方法中调用，将其初始化了即可。

2.6　习　　题

一、填空题

(1) 一般的 URL 都是由协议、主机名、端口号、路径、__________和信息片断等部分组成。

(2) URLRouting 颠覆了 URL 和文件系统一对一的关系，它可以将 URL 直接映射到一个 Controller(控制器)中的__________上。

(3) URLRouting 位于__________命名空间。

(4) Route 类的 Constraints 属性是以__________表示的字符串。

二、选择题

(1) 在 Route 类中，下列属性中的__________属性用于配置路由的路径信息。

A. Constraints　　B. Defaults
C. RouteHandler　　D. URL

(2) RouteTable 类的 Routes 属性是一个__________类型的属性。

A. RouteCollection　　B. List
C. Routes　　D. RouteTable

(3) 在 Route 类的 URL 属性中的占位符里，约定__________代表接收请求的控制器类。

A. controller　　B. control
C. Controllers　　D. action

(4) 在 Route 类的 URL 属性中的{action}参数所映射的方法必须是__________的方法，否则将访问不到。

A. public　　B. protected
C. Friend　　D. private

三、上机练习

上机练习： 自定义一个路由规则。

自定义一个路由规则，该路由规则可以匹配如下 URL:

```
News/2010/11/3/asp.net
News/2010/8/31/mvc/3
```

这里要求第一部分必须为 News，第二、第三和第四部分分别是日期的年、月和日，最后面的所有字符为一部分。

请配置出完整的 URLRouting 规则。



第 3 章　Controller 及 Action

内容摘要

本书前面章节讲解了 ASP.NET MVC 框架的基本概念和 URLRouting，为本章的讲解打下了坚实的基础。下面将更详细地讨论 MVC 架构中的一个核心元素——控制器。

控制器可以理解为用户接收请求，将模型与视图匹配在一起，共同完成用户的请求。划分控制器的作用也很明显，它清楚地告诉你，它就是一个分发器，选择什么样的模型，选择什么样的视图，可以完成什么样的用户请求。控制器并不做任何数据处理。例如，用户单击一个链接，控制层接收请求后，并不处理业务信息，它只把用户的信息传递给模型。告诉模型做什么，然后选择符合要求的视图返回给用户。因此，一个模型可能对应多个视图，一个视图可能对应多个模型。

模型、视图与控制器的分离，使得一个模型可以具有多个显示视图。如果用户通过某个视图的控制器改变了模型的数据，那么所有其他依赖于这些数据的视图都应反映这些变化。无论何时发生了何种数据变化，控制器都会将变化通知所有的视图，使显示更新。

MVC 模型中的控制器(Controller)主要负责响应用户的输入，可以修改模型(Model)以响应用户的输入。这样，MVC 模式中的控制器主要关注的是应用程序中的流动，处理引入的数据，并提供输出到相关视图(View)的数据。

本章主要为大家讲解 Controller 的实现原理。

学习目标

- 了解并掌握控制器的创建
- 掌握控制器类和动作
- 掌握 Response.Write()的用法
- 掌握 ActionResult 类的使用
- 了解并掌握映射参数
- 掌握 RedirectToAction 方法的使用

3.1 创建 Controller

控制器是 MVC 应用程序的构造函数，严谨地勾画了用户、模型对象以及视图的交互作用。它负责响应用户输入，操纵适当的模型对象，然后选择适当的视图来显示给用户以响应最初的输入。

MVC 控制器负责响应对 ASP.NET MVC 网站发起的请求。每个浏览器请求都将被映射到一个专门的控制器。本节就为大家讲解如何创建一个 Controller。

 **视频教学：**光盘/videos/03/3.1 创建 Controller　　长度：6 分钟

3.1.1 基础知识——Controller 的要求

控制器主要负责接受和解释输入，并更新任何需要的数据类(模型)，然后通知用户进行了修改或程序更新(本书将在第 5 章详细讨论视图)。

控制器继承 System.Web.Mvc.Controller 类，当新建一个 ASP.NET MVC 2 Web 应用程序时，系统会自动生成控制器，包含在 Controllers 文件夹中，控制器名称以 Controller 结尾。例如，XyzController 的名称即为 Xyz，一般首字母要大写。创建了应用程序之后，【解决方案资源管理器】窗格下面就会出现控制器所在的文件夹 Controllers，如图 3-1 所示。

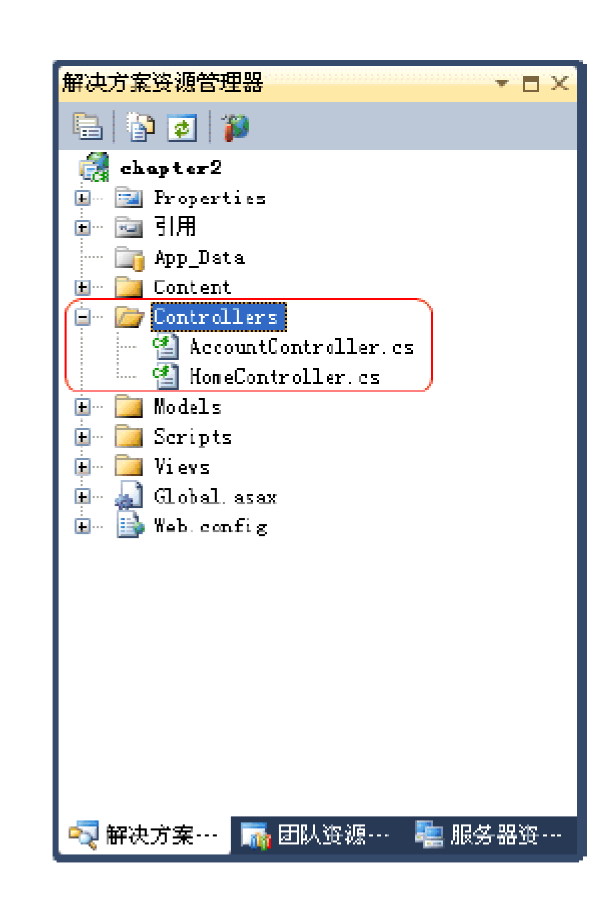


图 3-1 控制器

3.1.2 实例描述

小鸟之所以能够在天空中飞翔，是因为它有翅膀；人们之所以能够从河的这边到达河的那边，是因为河上搭了一座桥；卫星之所以能够被成功地发射，是因为有运载火箭的支持。同理，用户信息之所以能够显示在页面中，是因为有控制器的存在。也就是说用户信息是通过控制器来响应的，以下为大家讲解如何创建控制器。

3.1.3 实例应用

【例 3-1】创建 Controller。

(1) 新建 ASP.NET MVC 2 Web 应用程序，命名为 chapter3。默认生成两个控制器，即 Account 和 Home。如果我们要新建一个控制器，最简单的方法就是在【解决方案资源管理器】的 Controllers 文件夹上右击鼠标。并依次选择【添加】|【控制器】命令，如图 3-2 所示。

弹出【添加控制器】对话框，如图 3-3 所示。如果需修改名字，可在此对话框中进行修改。

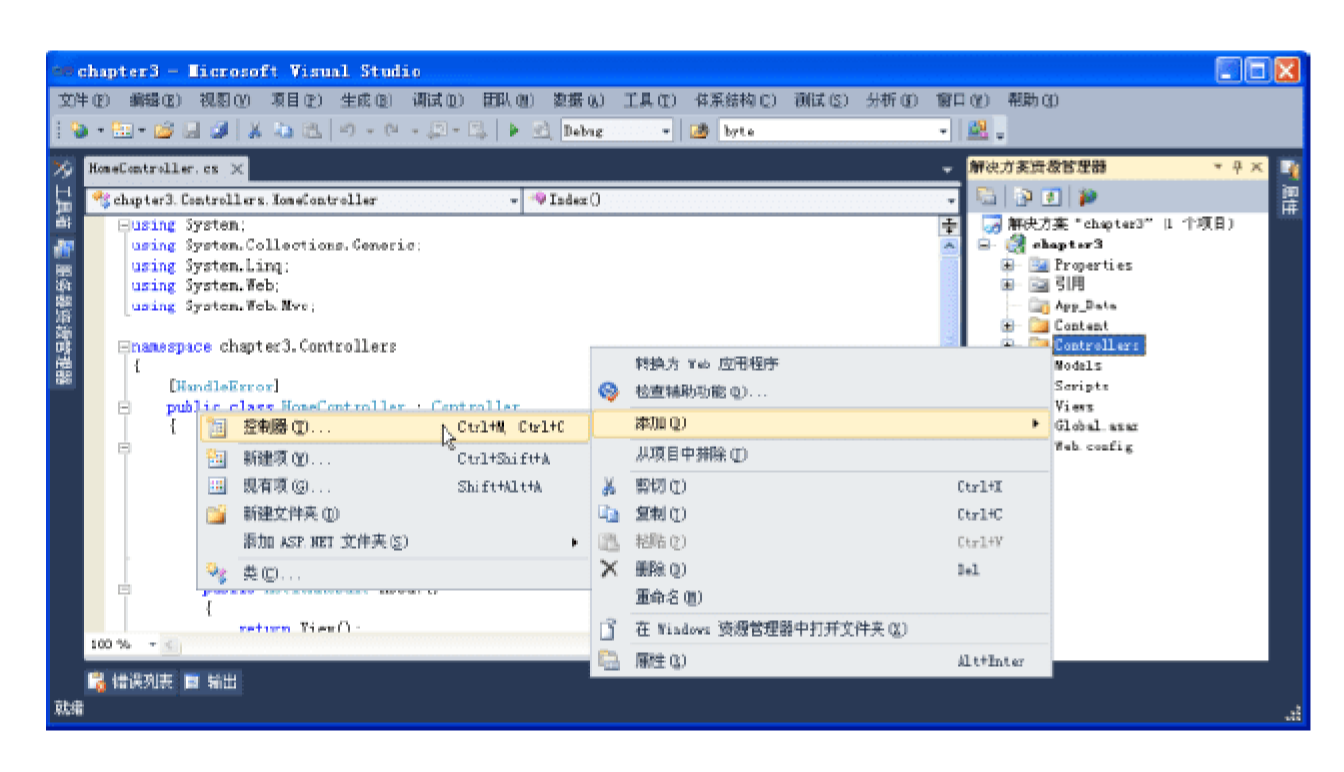


图 3-2　新建控制器

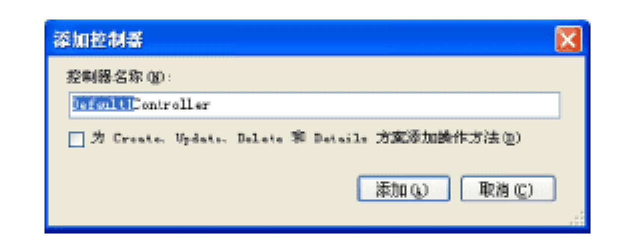


图 3-3　【添加控制器】对话框

(2) 控制器里默认有两个动作方法——Index()和 About()，它们的返回值都为 ActionResult，默认生成的代码如下：

```
public ActionResult Index()
{
    ViewData["Message"] = "欢迎使用 ASP.NET MVC!";
    return View();
}

public ActionResult About()
{
    return View();
}
```

3.1.4　运行结果

运行程序，默认加载 Home 控制器，它将默认寻找 Index 视图并显示出来，效果如图 3-4 所示。



图 3-4　控制器

3.1.5　实例分析



源码解析

控制器是 ASP.NET MVC 2 Web 应用程序自带的，当然也可以添加控制器。那么如何添加

控制器呢？在以上案例中已经讲过其操作步骤了。

关于项目创建时自带的控制器，也就是 Home 和 Account 控制器，当运行程序的时候，会默认加载 Home 控制器，并显示 Index 视图，即默认的 URL 地址为~/home/index。

3.2　获取产品列表

在 Web 项目开发过程中，有很多种框架可供选择，例如.NET Framework 框架和 MVC 框架等。

本节将以获取产品列表为例，为读者讲解 MVC 框架中的控制器类和动作。

**视频教学：**光盘/videos/03/3.2　获取产品列表　　长度：8 分钟

3.2.1　基础知识——控制器类和动作

编写控制器的标准方法是让控制器继承 System.Web.Controller 的抽象基类，它可以实现 Controller 的基类。因为控制器类向源自基类的控制器提供很多用于响应对 ASP.NET MVC 网站所进行的 HTTP 请求的方法，所以它是用来为所有控制器的基类服务的。

源自控制器的类的所有公有方法都将成为动作方法，它们通过 HTTP 请求来调用。但与 Execute 的单块式实现不同，用户可以在控制器里创建多个动作方法，每个方法都将响应具体的用户输入。

动作是控制器的一个方法，当你在浏览器地址栏中输入某一特定的 URL 时，将会调用这个方法。举个例子，假设你对下面这个 URL 发出请求：

```
http://localhost/Product/Index/3
```

在本例中，Index()方法在 ProductController 类上被调用。Index()方法是控制器动作的一个实例。

一个控制器动作必须是控制器类的一个公共方法。控制器动作还要满足一些额外的需求。作为控制器动作来使用的方法不能够重载。另外，控制器动作不能为静态方法。除此之外，你可以将任何方法作为控制器动作来使用。

3.2.2　实例描述

“海纳百川，有容乃大”，这句话是用来形容一个人心胸像大海一样宽阔。

获取产品列表，可以从一个泛型集合里面获取产品信息。这里的泛型集合，就相当于大海一样，可以容纳很多条产品信息。下面就用控制器动作来实现产品信息的添加与读取。

3.2.3　实例应用

【例 3-2】获取产品列表。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 chapter3。在 Models 文件夹下新建一个包含产品数据的类，名字为 Products，产品数据代码如下：

```
public class Products
{
    /// <summary>
    /// 产品编号
    /// </summary>
    public int ProId { get; set; }

    /// <summary>
    /// 产品名称
    /// </summary>
    public string ProductName { get; set; }

    /// <summary>
    /// 产品价格
    /// </summary>
    public double ProductPrice { get; set; }
}
```

(2) 如上所述，Products 相当于一个仓库的名字，现在仓库里面没有产品，我们就要往里面添加商品。实例化一个泛型 List<Models.Products>，名字为 Producct s。然后用 Add()方法向 Products 里添加数据，代码如下：

```
public ActionResult Index()
{
     List<Models.Products> products = new List<Models.Products>();
     products.Add(new Models.Products { ProId = 1, ProductName = "数码相机",
     ProductPrice = 2000 });
     products.Add(new Models.Products { ProId = 2, ProductName = "冰箱",
     ProductPrice = 2000 });
     products.Add(new Models.Products { ProId = 3, ProductName = "笔记本",
     ProductPrice = 2000 });
     products.Add(new Models.Products { ProId = 4, ProductName = "液晶显示器",
     ProductPrice = 2000 });
     products.Add(new Models.Products { ProId = 5, ProductName = "音响",
     ProductPrice = 2000 });
     products.Add(new Models.Products { ProId = 6, ProductName = "电脑桌",
     ProductPrice = 2000 });
     return View(products);
}
```

(3) 在视图中获取动作方法传递过来的数据。在 ASP.NET MVC 框架中添加视图的时候，可以自动与 Model 类关联，如图 3-5 所示。【视图数据类】选为 Products 类，【视图内容】选为 List，并选择你想要的母版页。在这里，要注意将生成的代码放在指定的位置。

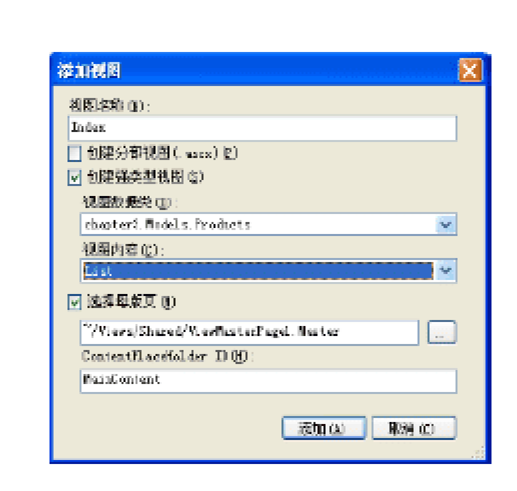


图 3-5　【添加视图】对话框

3.2.4　运行结果

运行程序，URL 默认地址为 http://localhost:2377/，这个时候 Home 控制器会默认寻找 Index()动作方法，也就是说实际的默认地址为 http://localhost:2377/Home/Index，如图 3-6 和图 3-7 所示。



图 3-6　默认地址



图 3-7　实际默认地址

运行程序，效果如图 3-8 所示。



图 3-8　获取产品列表

3.2.5　实例分析


源码解析

以 Home 控制器为例，运行程序之后，它将默认寻找 Index()动作方法，并通过该动作方法找到对应的视图。

当然，我们也可以用其他控制器去寻找对应的视图。在该案例中，用控制器动作保存了一个泛型集合 Products，要想让这些数据显示在视图中，最简单的方法就是在添加动作方法对应的视图时，指定 Model 类以及要显示的数据形式。

3.3　没有 MV 的 ASP.NET MVC

MVC 框架分为 3 个部分，分别为 Model、View 和 Controller。通常，MVC 框架的 3 个部分是共同使用的，使得代码逻辑更加规范。现在我们可以只用控制器来实现页面的展示效果。


视频教学：光盘/videos/03/3.3　没有 MV 的 ASP.NET MVC　　长度：4 分钟

3.3.1　基础知识——Response.Write 方法

在 Response 中 Write 方法是最常用的方法，该方法可以向浏览器动态输出信息。任何类型数据，只要是 ASP.NET 中合法的数据类型，都可以用 Response.Write 方式来显示。

3.3.2　实例描述

所谓没有 MV 的 MVC，就是只使用控制器层就能够显示所设计的页面。下面的案例将为大家实现这样的效果。

3.3.3　实例应用

【例 3-3】没有 MV 的 ASP.NET MVC。

创建一个没有返回值的动作方法，名称是 WriteHtml()。用 Response.Write()方法来输出静态页面，代码如下：

```
Response.Write("<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.1//EN'
'http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd'>");
Response.Write("<html>");
Response.Write("<head>");
Response.Write("<meta http-equiv='content-type' content='text/html;
charset=iso-8859-1'/>");
Response.Write("<meta name='description' content='description'/>");
Response.Write("<meta name='keywords' content='keywords'/>");
Response.Write("<meta name='author' content='author'/>");
Response.Write("<link rel='stylesheet' type='text/css'
href='../../Content/default.css' media='screen'/>");
Response.Write("<title>艺术人生</title>");
Response.Write("</head>");
Response.Write("<body>");
Response.Write("<div class='header'>");
Response.Write("<h1>艺术人生</h1>");
Response.Write("</div>");
Response.Write("<div class='navigation'>");
Response.Write("<a href='index.html'>艺术成就</a>");
Response.Write("<a href='index.html'>和平助人</a>");
Response.Write("<a href='index.html'>完美开朗</a>");
Response.Write("<a href='index.html'>艺术成就</a>");
Response.Write("<a href='index.html'>思想智慧</a>");
Response.Write("<div class='clearer'><span></span></div>");
Response.Write("</div>");
Response.Write("<div class='container'>");
Response.Write("<div class='content'>");
Response.Write("<h1>中国古代的思想家认为:</h1>");
Response.Write("<p>孟子曰：人之初，性本善<a href='index.html'>荀子曰：人之初，性本
恶，善者伪也</a> 告子说：人之初，性本无向，犹如水，决之东则东，决之西则西。</p>");
Response.Write("<cite>人性是罪恶的，来到这世界是为了赎罪，可我们却在越陷越深。马克思认
为：人是社会关系的产物。</cite>");
Response.Write("<p>所有这些问题都在于想对人性整体作一个说明，这些认识都是基于人生命不存
在生命层次区别的不同。</p>");
Response.Write("<p>其实人是一个三个生命层次即灵、魂、体的复合体，三个生命层次的属性可以
相同,也可以不相同 ……");
Response.Write("</p>");
Response.Write("<div class='divider'></div>");
Response.Write("<h1>人的灵魂或精神</h1>");
Response.Write("<p>人的灵魂或精神，是支配肉体和自我意识的主体，是对客观认识的逻辑体系，
是人的理性知识和智慧世界的主体。……</p>");
Response.Write("<ul>");
```

```
Response.Write("<li>平安</li>");
Response.Write("<li>喜乐</li>");
Response.Write("<li>仁爱</li>");
Response.Write("</ul>");
Response.Write("<p>例如，一个小孩的魂或精神就最容易受伤害，一个较重的适当(或不适当)的批评，就可以让小孩在某些方面完全否定自己在积极方面已有的成长，……</p>");
Response.Write("<div class='divider'></div>");
Response.Write("<h1>人的知识逻辑体系</h1>");
Response.Write("<p>可见，积极方面的培养很不容易，可谓百年树人。但消极方面的毁灭可以说是非常容易的一件事。……</p>");
Response.Write("<code>margin-bottom: 12px;font: normal 1.1em 'Lucida Sans Unicode',serif;background: url(img/quote.gif) no-repeat;padding-left: 28px;color: #555;</code>");
Response.Write("<p>可见，积极方面的培养很不容易，可谓百年树人。但消极方面的毁灭可以说是非常容易的一件事。</p>");
Response.Write("</div>");
Response.Write("<div class='footer'>");
Response.Write("&copy; 2006 <a href='index.html'>Website</a>. Valid <a href='http://jigsaw.w3.org/css-validator/check/referer'>CSS</a> &amp; <a href='http://validator.w3.org/check?uri=referer'>XHTML</a>. Design by <a href='http://arcsin.se'>Arcsin</a>");
Response.Write("</div>");
Response.Write("</div>");
Response.Write("</body>");
Response.Write("</html>");
```

3.3.4 运行结果

运行程序，效果如图 3-9 所示。

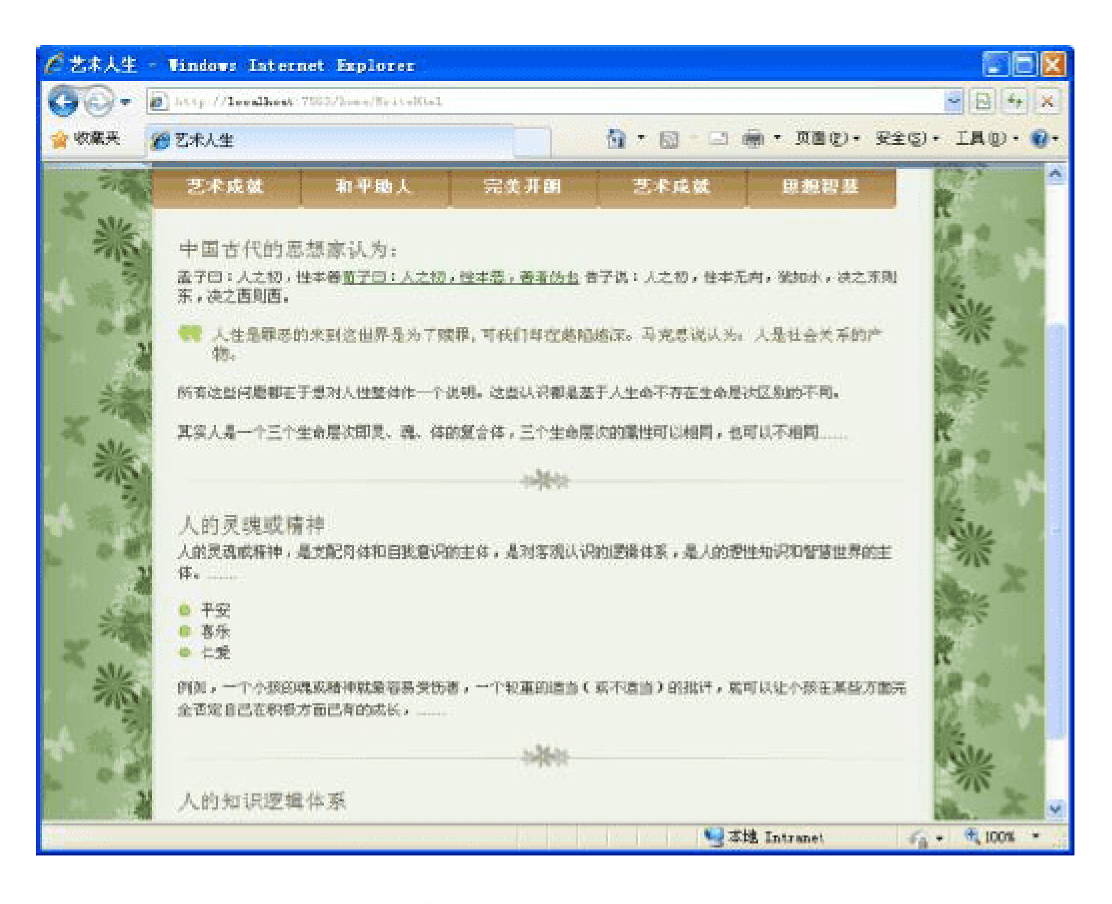


图 3-9　没有 MV 的 ASP.NET MVC

3.3.5 实例分析



源码解析

Response.Write()方法有很多种用途。例如，输出 JavaScript 脚本，输出 HTML 代码等。

由于 Response.Write()方法可以直接输出 HTML 代码，那么我们就可以直接在控制器中输出设计好的 HTML 页面，而不用管视图和模型中的代码应该写什么，只需要通过控制器就可以实现一个很漂亮的网页。

3.4　提交购物车到订单

提交购物车到订单，就是将购物车内的商品信息添加到订单中。本节主要讲解有关 ActionResult 类的知识，使读者对控制器有更深一层的认识。

**视频教学：**光盘/videos/03/3.4　提交购物车到订单(1)　长度：10 分钟
提交购物车到订单(2)　长度：5 分钟

3.4.1　基础知识——ActionResult 类

动作方法通过 Response.Write()将文本直接写到 HTTP 响应中。当然，虽然这是生成 HTTP 响应的一种有效的方法，但是这并不是最高效的方法，此外它还挫败了 ASP.NET 的一些比较整洁的功能，比如 Master Pages。

前面讲到，MVC 模式中控制器的用途是响应用户输入。在 ASP.NET MVC 中，动作方法是对用户输入响应的粒度单元。动作方法最终负责处理用户请求并输出显示给用户的响应，即通常的 HTML 脚本。

动作方法遵循的模式是完成任何要求的工作，并在最后返回继承自 ActionResult 抽象基类的类型的实例。

下面来看 ActionResult 抽象基类的源代码。

```
    //封装一个操作方法的结果并用于代表该操作方法执行框架级操作
    public abstract class ActionResult
 {
      //初始化 System.Web.Mvc.ActionResult 类的新实例
      protected ActionResult();
      //通过从 System.Web.Mvc.ActionResult 类继承的自定义类型，启用对操作方法结果的处理
      //参数 context 用于执行结果的上下文。上下文信息包括控制器、HTTP 内容、请求上下文和
路由数据。
      public abstract void ExecuteResult(ControllerContext context);
  }
```



该类包含了一个构造函数和一个方法，方法就是 ExecuteResult。动作结果代表了动作方法想要架构为我们完成的命令。

一般来说，动作结果处理的是架构层次上的工作，而动作方法处理的是应用程序的逻辑。例如，当请求显示一列产品列表时，动作方法将查询数据库，并将合适的产品列表放在一起显示出来，其中可能需要基于应用程序中的业务规则来完成某种过滤。至此，动作方法完全将注意力集中在应用程序的逻辑上。

然而，一旦方法为显示产品列表做好准备之后，可能不希望动作方法中的代码直接处理架

构层次上的信息管道，例如将产品清单写入响应中。可能会有一个自定义的模板，它知道如何将产品集合格式化成 HTML。最好不要把信息封装在动作方法中。

可供使用的一项技术是让动作方法实例化 ViewResult(继承自 ActionResult)的一个实例，并提供数据给该实例，然后返回该实例。此时，动作方法完成其工作，而动作调用者将调用关于 ViewResult 实例的 ExecuteResult 方法，这将完成剩余的工作；这里给出了代码可能的情况。

```
public ActionResult ListProducts()
{
    IList<Product> products=SomeRespository.GetProducts();
    ViewDate.Model=product;
    return new ViewResult(ViewDate=this.ViewDate);
}
```

在实践中，可能永远不会直接看到类似的实例化 ActionResult 实例的代码。取而代之，将使用 Controller 类中的一个辅助方法，例如下面的 View 方法：

```
public ActionResult ListProducts()
{
   IList<Product> products=SomeRespository.GetProducts();
   return View(products);
}
```

1. 动作结果的类型

ASP.NET MVC 包括执行常见任务的 ActionResult 类型。

1) EmptyResult

顾名思义，这一结果用来指明架构不做任何事情，这里遵循的常见设计模式叫做 Null Object 模式，它将通过一个实例来替换空的引用。在该实例中，ExecuteResult 方法具有一个空的实例。

2) ContentResult

这一结果将其指定的内容(通过 Content 属性指定)写入响应中。此外，这个类还支持指定内容的编码(通过 ContentEncoding 属性指定)以及内容类型(通过 ContentType 属性指定)。

如果没有指定编码，那么就使用当前 HttpResponse 实例的内容编码。HttpResponse 的默认编码是在 web.config 的全局化元素中指定的。

同样，如果没有指定内容类型，则使用当前 HttpResponse 实例上的内容类型设置。HttpResponse 默认的内容类型是 text/html。

3) FileResult

除了用于将二进制内容(例如，磁盘上的 Microsoft Word 文档或来自 SQL Server 中 blob 列的数据)写入响应中之外，该结果非常类似于 ContentResult。设置结果上的 FileDownloadName 属性并设置 Content-Disposition 题头的适当值，会导致一个文件下载对话框出现。

FileResult 是一个抽象基类，用于如下 3 个不同的文件结果类型：

- FilePathResult
- FileContentResult
- FileStreamResult

4) JsonResult

该结果使用 JavaScriptSerializer 类来将其内容(通过 Data 属性指定)串行化为 JSON(JavaScript Object Notation)格式。对于简单的 Ajax 情形，如果需要一个动作方法来以一

种易于为 JavaScript 消费的格式返回数据，那么这一结果将非常有用。

与 ContentResult 一样，JsonResult 的内容编码和内容类型都可以通过属性来设置。唯一的区别在于默认的 ContentType 是 application/json，而不是该结果对应的 text/html。



JsonResult 串行化整个对象图表。因此，如果提供一个 ProductCategory 对象(含有 20 个 Product 实例的集合)，那么还将串行化每个 Product 实例并将其包含到发送给响应的 JSON 中。现在，假设每个 Product 都含有一个包含 20 个 Order 实例的 Orders 集合，那么可以想象，JSON 响应可能迅速膨胀。

5) JavaScriptResult

JavaScriptResult 用来在客户端执行来自服务器的 JavaScript。例如，在使用内置的 Ajax 辅助方法发送给动作方法的请求时，方法可能只是返回一些 JavaScript，它将在到达客户端时立刻执行：

```
public ActionResult DoSomething(){
    script s="$('#some-div').html('Updated!')";
    return JavaScript(s);
}
```

这里假设引用了 Ajax 和 jQuery，将通过如下代码来调用：

```
<%=Ajax.ActionLink("click","DoSomething",new AjaxOption()) %>
<div id="some-div"></div>
```

6) RedirectResult

该结果将执行重新指向指定 URL 的 HTTP(通过 Url 属性设置)。从内部讲，该结果调用 HTTPResponse.Redirect 方法，将 HTTP 状态码设置为 HTTP/1.1 302 Object Moved，导致浏览器立刻发送一个对指定 URL 的新请求。

7) RedirectToRouteResult

执行 HTTP 重定向与执行 RedirectResult 的方式一样，不同的是，没有直接指定一个 URL，这一结果使用路由选择的 API 来确定重定向的 URL。

其中，有两个便捷的方法 RedirectToRoute 和 RedirectToAction，这两个方法将返回该类型的结果。

8) ViewResult

该结果调用 IViewEnging 实例的 FindView 方法，返回 IView 的一个实例。随后，ViewResult 调用 IView 实例上的 Render 方法，它将呈现响应的输出。一般来讲，这将把指定的视图数据(即动作方法准备显示在视图中的数据)融入一个模板中(该模板将对被显示的数据进行格式化)。

9) PartialViewResult

除了调用 FindPartialView 方法(而不是 FindView)来定位视图之外，该结果的工作方式与 ViewResult 的工作方式完全相同。它用来呈现局部视图(如 ViewUserControl)，而且在局部更新的时候，它起到了关键的作用。

2．动作结果的辅助方法

如果在默认的 ASP.NET MVC 项目模板中仔细查看默认的控制器动作，就会发现动作方法实际上没有实例化 ViewResult。例如，下面给出了 About 方法的代码：

```
public ActionResult About()
{
    return View();
}
```



它只是返回对 View 方法的调用结果。控制器类包含几个返回 ActionResult 实例的便捷方法，这些方法的目的是以更可读和更公开的方式来实现动作方法。这里没有创建动作结果的新实例，更为常见的是只返回其中一个便捷方法的结果。

动作结果的辅助方法通常是按照方法返回的动作结果类型来命名的，并使用 Result 作为后缀。因此，View 方法将返回 ViewResult 的一个实例。类似地，Json 方法返回 JsonResult 的一个实例。只有一个例外的情形，即 RedirectToAction 方法将返回 RedirectToRoute 的一个实例。表 3-1 列出了现有的方法及其返回的类型。

表 3-1　动作结果的辅助方法

方　法	说　明
Redirect()	返回一个 RedirectResult，将用户重新指向适当的 URL 地址
RedirectToAction()	返回一个 RedirectToRouteResult，将用户重新指向一个使用了所提供的路由值的动作
RedirectToRoute()	返回一个 RedirectToRouteResult，将用户重新指向与指定路由值匹配的 URL 地址
View()	返回一个 ViewResult，将视图呈现给响应
PartialView()	返回一个 PartialViewResult，将局部视图呈现给响应
Content()	返回一个 ContentResult，将指定的内容(字符串)写入响应中
File()	返回一个继承自 FileResult 的类，将二进制的内容写入响应中
Json()	返回一个 ContentResult，它包含了来自将对象串行化到 JSON 中的输出
JavaScript()	返回一个含有 JavaScript 代码的 JavaScriptResult，当返回到客户端时，立刻执行该代码

3. 隐式的动作结果

对于 ASP.NET MVC 以及通常的软件开发而言，不变的目标是让代码执行的目的尽可能地清晰。曾经有一段时间，让一个非常简单的动作方法只用于返回一个单一片段的数据。在这种情况下，让动作方法的签名来反映其返回的信息是非常有用的。

为了着重指出这一点，请看下面的返回值类型为 double 的动作方法。现在将距离计算的结果直接写入响应中，采用代表两点的参数(x1，y1)和(x2，y2)，返回一个代表两点距离的 double 型的值，实现代码如下：

```
public double Distance(int x1, int y1, int x2, int y2)
{
    double x = Math.Pow(x2-x1,2);
    double y = Math.Pow(y2-y1,2);
    return Math.Sqrt(x+y);
}
```



返回的类型是 double，而并不是继承自 ActionResult 的类型，这是可以接受的。当 ASP.NET MVC 调用该方法并发现返回的类型并不是一个 ActionResult 时，它将自动实例化一个含有动作方法结果的 ContentResult，并像 ActionResult 一样从内部使用它。

隐式的动作结果可能的返回值如表 3-2 所示。

表 3-2　隐式的动作结果

返回的值	说　明
空值	动作调用者通过一个 EmptyResult 的实例来替换空的结果，它遵循的是 Null Object Pattern。因此，实现者编写自定义动作过滤器而不必担心产生空的动作结果
Void	动作调用者将动作方法当作其返回空值来处理，因此返回了一个 EmptyResult
对象(除了 ActionResult 之外的任何对象)	动作调用者使用对象上的 InvariantCulture 来调用 ToString，并将结果字符串封装到 ContentResult 实例中

3.4.2　实例描述

随着互联网的快速发展，一部分消费者开始到网上购物，这样省力省时又省事。以前，我做过一个电子商务系统，客户注重的是购物车这一模块，这不仅涉及资金流动问题，也涉及消费者的使用感受。比如，用户想要购买的商品没货了，那么就可以将购物车中的商品提交到订单列表中，方便卖家对产品进行及时采购。

3.4.3　实例应用

【例 3-4】提交购物车到订单。

(1) 创建一个 ASP.NET MVC 应用程序，名字为 Shopping。

(2) 在 Models 文件夹下添加一个 ProductName 类，用于记录购物车中的产品信息。代码如下：

```
public class Product
{
    /// <summary>
    /// 产品编号
    /// </summary>
    public int ProId { get; set; }
    /// <summary>
    /// 产品名称
    /// </summary>
    public string ProductName { get; set; }
    ///<summary>
    ///产品价格
    /// </summary>
    public double ProductPrice { get; set; }
    /// <summary>
    /// 产品数量
    /// </summary>
    public int Number { get; set; }
}
```

(3) 添加一个控制器，名称为 GouWuCheController。在控制器里添加一个带有返回值的方法，名称为 AddProducts，表示要向购物车中添加商品，返回一个泛型集合。向购物车中添加 5

条数据，代码如下：

```
public List<Models.Product> AddProducts()
{
   List<Models.Product> products = new List<Models.Product>();
   products.Add(new Models.Product { ProId = 1, ProductName = "筷子",
   ProductPrice = 5, Number = 3 });
   products.Add(new Models.Product { ProId = 1, ProductName = "圆珠笔",
   ProductPrice = 5, Number = 3 });
   products.Add(new Models.Product { ProId = 1, ProductName = "笔记本",
   ProductPrice = 5, Number = 3 });
   products.Add(new Models.Product { ProId = 1, ProductName = "吹风机",
   ProductPrice = 50, Number = 1 });
   products.Add(new Models.Product { ProId = 1, ProductName = "发夹",
   ProductPrice = 5, Number = 3 });
   return products;
}
```

(4) 这些数据要提交到订单视图中，就先添加一个订单控制器 OrderController，并添加对应的视图 Index.aspx。创建强类型视图，连接对应的模型 Product，并选择以列表显示，如图 3-10 所示。

图 3-10　添加视图

(5) 将购物车中的商品信息提交到订单中，代码如下：

```
 public ActionResult Index()
 {
     List<Models.Product> products = AddProducts();
     return View("~/Views/Orders/Index.aspx", products);
 }
```

3.4.4　运行结果

运行程序，在地址栏中添加字符串 gouwuche/index 并回车，效果如图 3-11 所示。

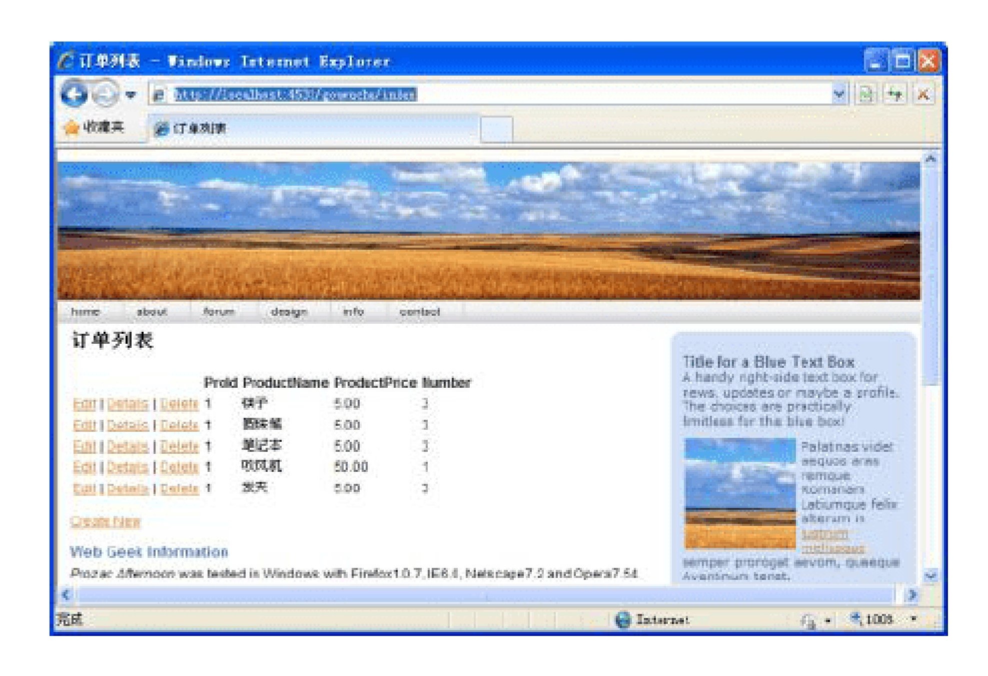


图 3-11　显示提交购物车到订单

3.4.5　实例分析



源码解析

在该案例中，Index()方法的返回值是 ActionResult 类。将产品信息从购物车提交到订单中，

这里创建了一个方法，用来向购物车中添加产品，返回值为泛型类。将数据传到订单页面，这里使用了 View 方法，代码如下：

```
return View("~/Views/Orders/Index.aspx", products);
```

其中，products 表示传递的泛型集合。

3.5 提交用户信息

在这个互联网的时代，不管是老人，还是小孩，都有自己的网名。当然，一个人可以有很多个网名，比如说我在网上看到一个很不错的论坛，想成为会员，我就要注册我的信息。进入注册界面，填写用户信息并提交。OK，这样你就成为这个论坛的会员了，很简单的。

本小节将为大家讲解如何用 MVC 框架来实现提交用户信息的功能。


视频教学： 光盘/videos/03/3.5 提交用户信息　　长度：6 分钟

3.5.1 基础知识——映射参数

找到了动作方法，调用者就负责将值映射给方法的参数。表 3-3 详细说明了调用者查找参数值的地方。

一旦调用者映射了动作方法的每个参数值，它就准备好调用动作方法本身了。此时，调用者将构建一个与当前动作方法有关的过滤器列表，并以正确的顺序调用与动作方法绑定在一起的过滤器。

表 3-3 调用者查找参数值的地方

位　置	说　明
Request.Form 集合	这是提交的表单，包含名称/值对
路由数据	特别是位于当前 RequestContext.RouteData.RouteValues 中。路由数据取决于拥有一个已定义的路由，它可以将请求的 URL 映射到动作方法的参数中
Request.QueryString 集合	这是添加到 URL 后面的名称/值对

3.5.2 实例描述

在 ASP.NET MVC 中，要将 View 中的数据传递到控制器中，主要通过发送表单的方式来实现。下面这个案例主要实现提交用户信息的功能。

3.5.3 实例应用

【例 3-5】 提交用户信息。

(1) 创建一个 ASP.NET MVC 空应用程序，名称为 Userinfo。

(2) 首先建立一个页面，用来让用户输入用户名和密码。我们知道，在 ASP.NET MVC 中不能直接请求 ASPX 文件，任何请求都要通过 Controller。

在 Home 控制器下新建一个名为 Get_userinfo 的 Action 方法，用 Request.Form 的方法来获取用户信息。具体代码如下：

```
public ActionResult Get_userinfo()
{
    string username = Request.Form["username"];
    string password = Request.Form["pwd"];
    if (username == "admin" && password == "admin")
    {
        ViewData["Message"] = username + "---" + password;
        return View();
    }
    return View();
}
```

(3) Action 方法做完了，我们还需要视图。添加 Get_userinfo 视图，用 GET 方式提交用户名和密码。代码如下：

```
 <% using (Html.BeginForm("Get_userinfo", "Home",FormMethod.Get))
    { %>
 UserName: <%=Html.TextBox("username") %>
 Password: <%=Html.TextBox("pwd")%>
 <input type="submit" value="提交" id="btn_sub" />
<%} %>
```



其中 Html.BeginForm()方法的第一个参数表示对应的 Action 方法，第二个参数表示对应的控制器，第三个参数表示提交的方式。

3.5.4 运行结果

运行程序，效果如图 3-12 所示。

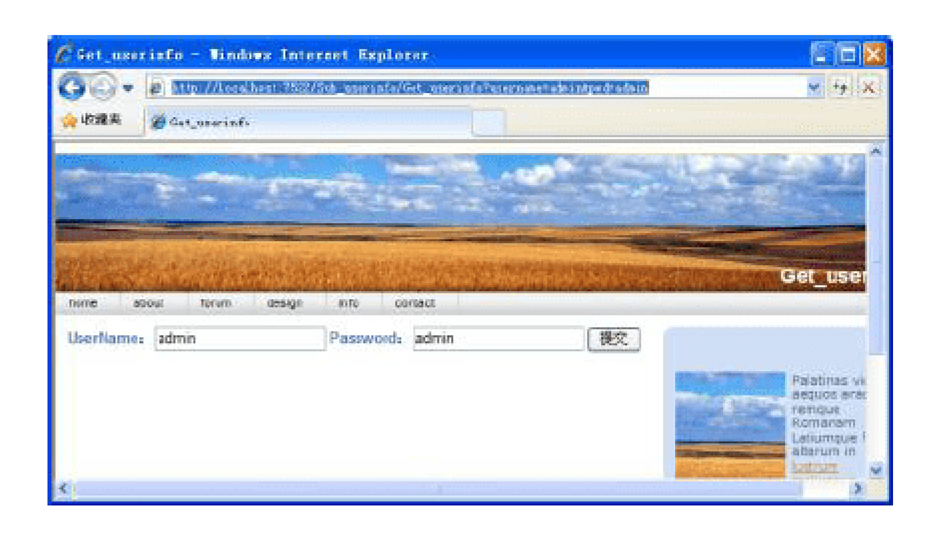


图 3-12 匿名提交用户信息

3.5.5 实例分析



源码解析

提交用户信息，首先就得确保用户输入信息在 Form 表单中。用户信息将包含在如下代码中：

```
Html.BeginForm("Get_userinfo", "Sub_userinfo",FormMethod.Get)
```

在这里设置了访问对象及方式。当 Form 表单结束之后，要自动释放。这里用到了 using()方法，也可以使用 Html.EndForm()方法最为 Html.BeginForm()方法的结束。

3.6　页面动作跳转

页面动作跳转，就是当用户单击某个链接或者按钮的时候，当前页面会跳转到另一个页面，这里所谓的“单击”就表示一个动作。本节就用 MVC 框架来实现页面动作跳转的功能。



视频教学： 光盘/videos/03/3.6　页面动作跳转　　长度：5 分钟

3.6.1　基础知识——RedirectToAction 方法

如果你想访问一个视图，那么你可以在控制器的动作方法中调用 View()方法。如果你想要将用户从一个控制器动作重定向到别处，那么你可以调用 RedirectToAction()动作跳转方法。

3.6.2　实例描述

在互联网上，我们常常可以通过图标或者按钮进入另一个页面，这种行为就叫做跳转。现在将从一个控制器的动作跳转到另一个控制器的动作中，在 ASP.NET MVC 框架中，这种行为就叫做动作跳转。接下来的这个案例，以页面动作跳转命名，为大家讲解动作跳转的实现过程。

3.6.3　实例应用

【例 3-6】页面动作跳转。

(1) 新建一个 ASP.NET MVC 应用程序，名称为 Action_tiaozhuan。

(2) 要实现页面的跳转，首先要添加两个控制器，分别为 PageOne 和 PageTwo，如图 3-13 和图 3-14 所示。

(3) 将以上两个控制器中的 Index()方法去掉，分别重新创建一个 Action 方法，PageOne 控制器的 Action 方法为 One()，PageTwo 控制器的 Action 方法为 Two()。要将 PageOne 的视图跳转到 PageTwo 的视图，先添加两个对应的视图。分别右击两个 Action 方法，弹出添加视图对话框，如图 3-15 和图 3-16 所示。

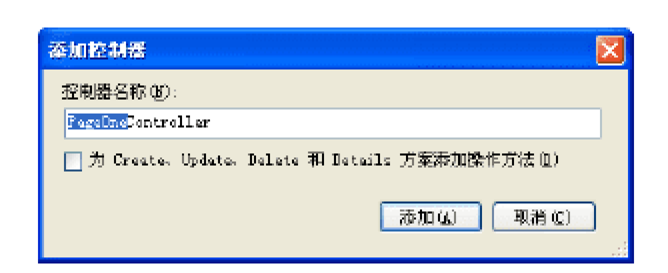


图 3-13　添加 PageOne 控制器

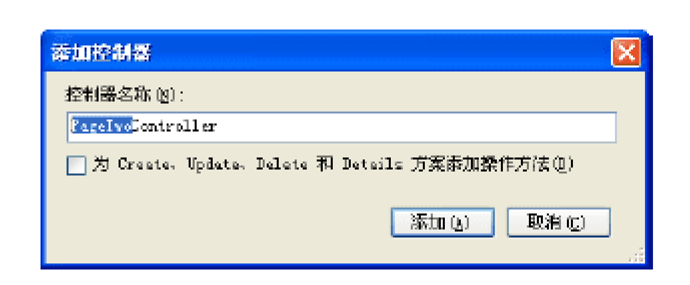
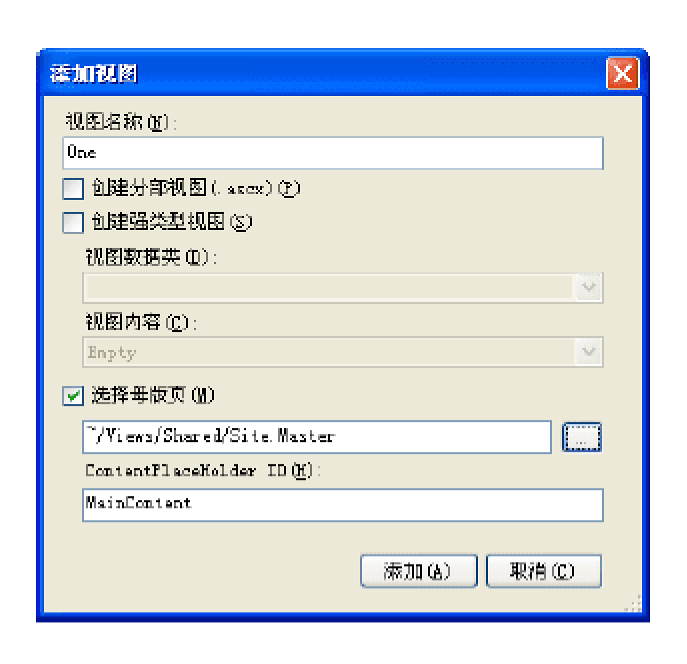


图 3-14　添加 PageTwo 控制器



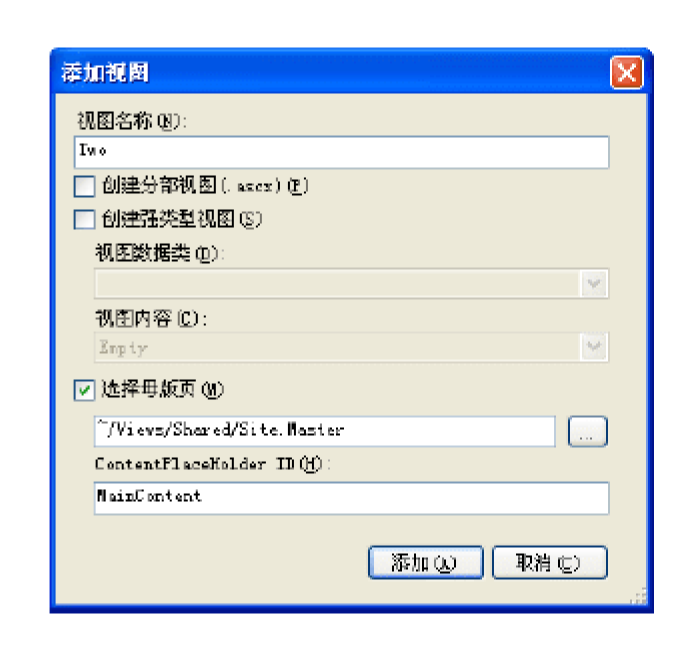




图 3-15　添加 One 视图　　　图 3-16　添加 Two 视图

(4) 分别为它们选择母版页，单击【选择母版页】选项组的□按钮，弹出【选择母版页】对话框，然后选择 ViewMasterPage1.Master 母版页，如图 3-17 所示。

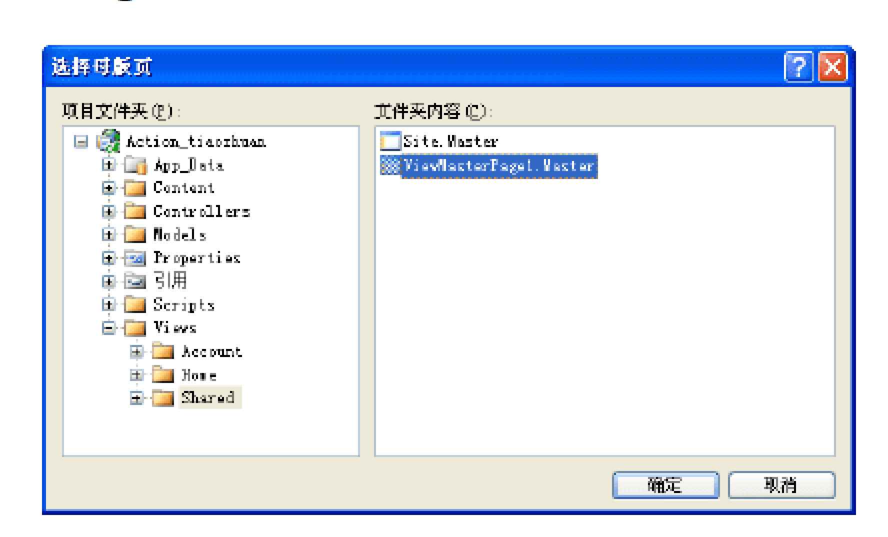


图 3-17　选择母版页

(5) 页面动作从 PageOne 控制器的 One 动作跳转到 PageTwo 控制器的 Two 动作。用 RedirectToAction()方法来跳转，代码如下：

```
public ActionResult One()
{
    return RedirectToAction("Two", "PageTwo");
}
```



这里的第一个参数表示将要跳转的目的页面的动作方法，第二个参数表示对应的控制器名称。

3.6.4　运行结果

运行程序，在地址栏里输入/pageone/one 并回车，URL 地址变为/pagetwo/two，如图 3-18 所示。

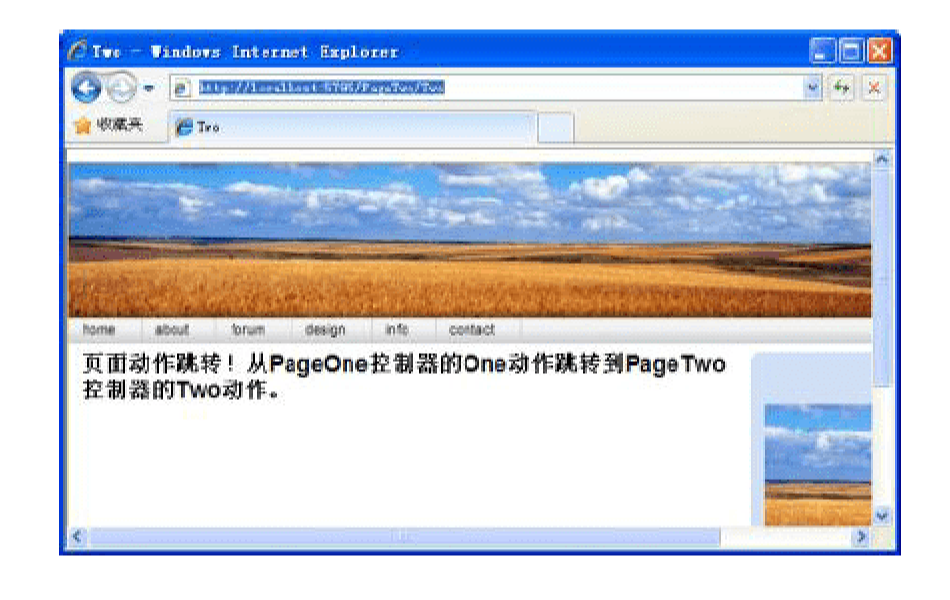


图 3-18　页面动作跳转

3.6.5　实例分析


源码解析

该案例主要讲述了 RedirectToAction 跳转方法，其实还有其他跳转方法，例如 Redirect 和 server.transfer 等。

RedirectToAction()方法有多个重载，该案例中用的是包含两个参数的重载，第一个参数是跳转的 Action 方法，第二个参数是控制器名称。代码如下：

```
return RedirectToAction("Two", "PageTwo");
```

3.7　常见问题解答

3.7.1　ASP.NET MVC 登录的问题


ASP.NET MVC 登录的问题。
网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

我写了一个登录的 POST 方法如下：

```
[HttpPost]
public ActionResult AdminLogin(string UserName, string Password)
{
    if (tbuserBll.ValidateUser(UserName, Password) == 0)
    {
        Session["userid"] = tbuserBll.GetUserIDbyUserName(UserName);
        Session["username"] = UserName;
        return RedirectToAction("Show", "Admin");
    }

    else if (tbuserBll.ValidateUser(UserName, Password) == 2)
    {
        Response.Write("<script>alert('该账号已被封停')</script>");
        return View("AdminLogin");
    }
    else
    {
        Response.Write("<script>alert('账号或密码错误')</script>");
        return View("AdminLogin");
    }
```

我想在母版页中显示出“欢迎您***用户”，怎么做？如果要在 show 页面调用 Session 的值，为什么非要用 return RedirectToAction("Show","Admin");而不能用 returnView("Show")。

先谢谢高手了，请讲具体点好吗，谢谢……

【解决办法】RedirectToAction 方法有 6 个重载，这个方法将会跳转到另一个 action 中去，两个 string 参数的重载方法指明：第一个参数是 action 的名字，第二个参数是 Controller 的名字。

3.7.2 Controller 如何返回 DataTable 给页面



ASP.NET MVC 中的 Controller 如何返回 DataTable 给页面？
网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

```
public class MyController : Controller
{
       //
       // GET: /My/

       public ActionResult Index()
       {
          ViewData["my"] = "this is my first mvc example!";
          return View();
       }

       public ActionResult GetData()
       {
          DataTable dtSource = new DataTable();
          dtSource.Columns.Add(new DataColumn("姓名",typeof(string)));
          dtSource.Columns.Add(new DataColumn("年龄", typeof(int)));

          dtSource.Rows.Add(new object[]{"张三",20});
          dtSource.Rows.Add(new object[] { "李四", 18 });

          return View();
       }
}
```

这是全部家当了，请大家指点!
【解决办法】在方法中这样写，把 DataTable 存到 ViewData 中。

```
ViewData["data"]=dtSource;
```

然后页面上会取出来。

3.7.3 Controller 中的变量问题



ASP.NET MVC 在 Controller 中的变量问题。
网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

在 Controller 定义了一个变量，这个变量有页面操作时返回一个值。

```
int pid=0;
public ActionResult searchByGategory()
{
     pid = Convert.ToInt32(Request.Form["list"]);
     return RedirectToAction("DictionaryContent");
}
```

当方法转到 DictionaryContent 时，pid 的值就变了，即重新变回 0。这种情况的原因是什么，应该怎么处理呢？

【解决办法】修改返回值的参数。

```
return RedirectToAction("DictionaryContent", new{pid=pid});
```

3.7.4　ASP.NET MVC 的传值问题



ASP.NET MVC 的传值问题。

网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

我刚学 MVC 框架，就是我想从甲页面传一个值到乙页面，怎么做？首先，我在 Index.aspx 视图里写了如下代码：

```
<%=Html.ActionLink("点我", "Happi", "Home", new { uname = "小蔡" })%>
```

HomeController.cs 里面的代码如下：

```
   public ActionResult Happi(String uname)
    {
          ViewData["haha"] = uname;
          return View();
     }
```

在 Happi.aspx 视图里的代码是：

```
<%=ViewData["haha"]%>
```

感觉这样应该没什么问题，可是我将鼠标在超链接上悬浮的时候，显示的是 http://localhost:6846/Home/Happi?Length=4 而不是 http://localhost:6846/Home/Happi?uname= BadBoy 当我在地址栏直接入 http://localhost:6846/Home/Happi?uname=BadBoy 却能够正确传值。

【解决办法】写的时候看着点参数名。

```
<%=Html.ActionLink("点我", "Happi", "Home", new { uname = "小蔡" })%>
<%=Html.ActionLink(显示名称, Action, 要传递的 routeData, html 属性)%>
```

这里有 4 个参数，第 3 个是 routeData，而第 4 个是 htmlAttribute，也就是你实际传的是 Home，所以 Length=4。如果要传递第 4 个参数，需要使用 5 个参数的重载，应这样写：

```
<%=Html.ActionLink("点我", "Happi", "Home", new { uname = "小蔡" },null)%>
```

这样就行了，或者使用不指定 Controller 的方法。

```
<%=Html.ActionLink("点我", "Happi", new { uname = "小蔡" })%>
```

3.8　习　　题

一、填空题

(1) 控制器名称必须以____________结尾。

(2) 一个控制器动作必须是控制器类的一个____________方法。

(3) MVC 控制器负责响应对________网站发起的请求。每个浏览器请求都被映射到一个________的控制器。

(4) 模型、视图与________的分离，使得一个模型可以具有________个显示视图。

(5) 调用者查找参数值的地方，包括________、路由数据和________。

二、选择题

(1) 下面________选项是动作结果的辅助方法。

A. show　　B. hide

C. ViewData　　D. View

(2) 以下________选项是 ASP.NET MVC 中动作跳转的方法。

A. Redirect　　B. RedirectToAction

C. Transfer　　D. Link

(3) 控制器涉及的内容不包含________。

A. 控制器(Controller)　　B. 控制器方法(Action)

C. 控制器结果(ActionResult)　　D. 视图(View)

(4) 下列________选项不属于动作结果类型。

A. RedirectResult　　B. FileInfo

C. JsonResult　　D. ContentResult

(5) 用于输出页面标签的方法是________。

A. Request.form　　B. Request.QueryString

C. Response.Write　　D. Response.WriteLine

三、上机练习

上机练习：将控制器中动作方法保存的数据显示在视图中。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序的项目 MvcApp，添加一个控制器，名字叫 ControlController。

(2) 创建一个新的母版页，页面设计自定义。

(3) 将控制器中动作方法设置的数据显示在视图中，类似效果如图 3-19 所示。

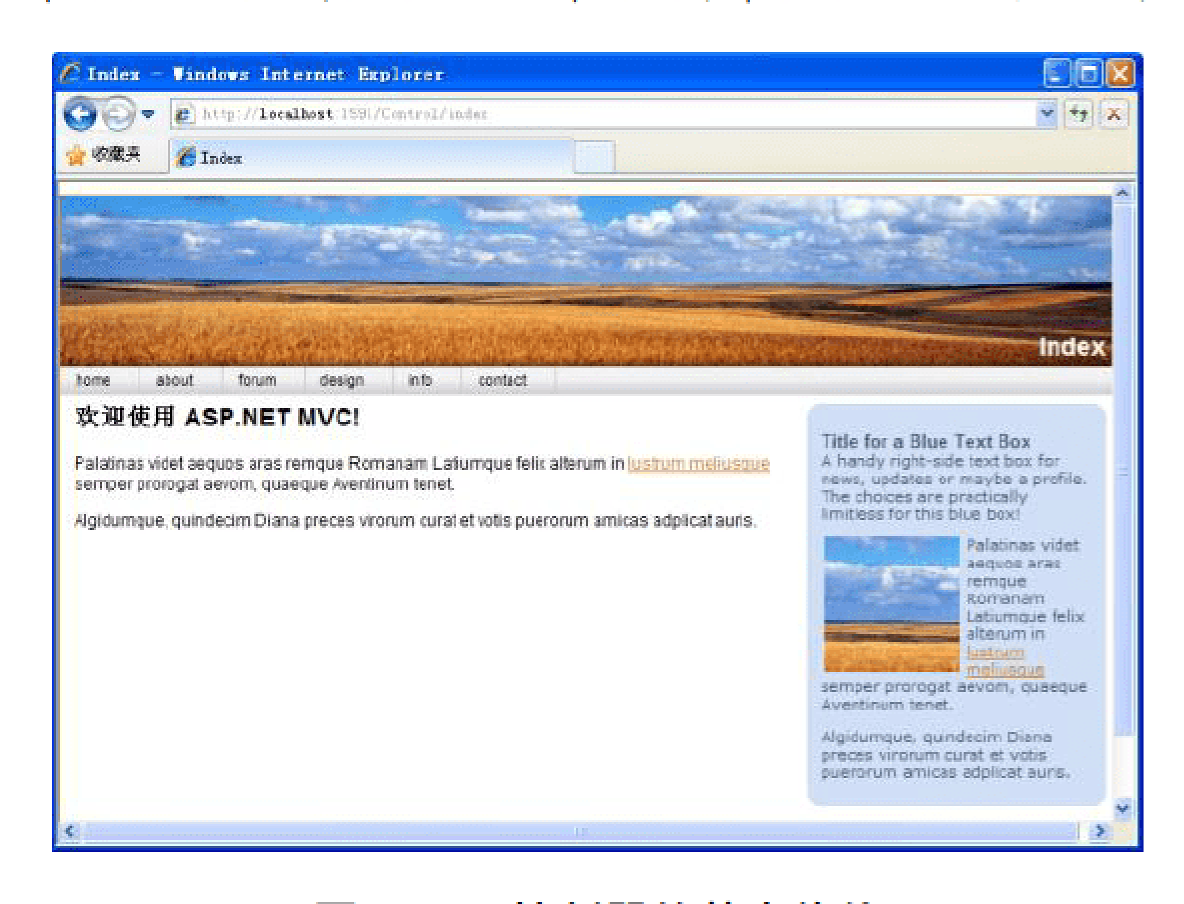


图 3-19　控制器的基本传值



第 4 章　Model

内容摘要

在使用软件的过程中，用户界面需要经常发生变化。采用 MVC 架构模式，可以让我们在满足界面要求的同时，使软件的计算模型独立于界面的构成。

MVC 模式是软件工程中的一种软件架构模式，它可以把软件系统分成 3 个基本部分，分别是：Model 模型、View 视图和 Controller 控制器。

本章就来详细讲解 Model 模型。主要对 Model 模型在 MVC 中所扮演的角色、重要性以及如何使用进行详细讲解。

学习目标

- 了解 Model 在 MVC 中所扮演的角色
- 了解 Model 的重要性
- 熟悉 Model 模型的使用

4.1 Model 简介

在 MVC 架构模型中，M 代表 Model 模型，它的职责就是处理数据和业务逻辑。应用程序被分成 3 个主要部分，每个部分都掌管着不同的任务。下面我们就来讲解 MVC 中的 M。

Model 用来存放独立且可以重复使用的组件，包括对数据来源(数据库)的访问，商业逻辑代码，并应用 View 做完整的切割，以便日后扩充或改变。

Model 模型是指运用于数据之上的数据规则和数据内容，它一般与应用程序中所要管理的对象对应。在软件系统中，任何事物都可以被抽象成对象，即可以对其以某种数据方式进行处理的数据模型。例如，应用程序中的用户信息是什么？其实，它们只是一堆必须按照对应规则处理的数据。

MVC 设计模式包括 Model 模型、View 视图和 Controller 控制器 3 个部分，首先需要实现的是 Model 部分。

4.2 Model 的重要性

我们先来谈一下三层架构中 Model 的作用。其实 Model 层的作用就是用来传参，但是如果有 20 个或者更多的参数怎么传呢？这就是 Model 的好处了，Model 基本可看作数据库中表的集合，通过 get 和 set 访问器，使其能够传递更多的值。

比如，student 表在 Model 中就会有这样的类声明属性，属性是和数据库中的字段一致的，那么不管你是 update 还是 insert，都可以直接实例化 Model 层的类给属性赋值。下面举个例子，学生表的 Model 层代码如下：

```
private string name;
public string Name
{
   get { return name; }
   set { name = value; }
}
private string sex;
public string Sex
{
   get { return sex; }
   set { sex = value; }
}
private int age;
public int Age1
{
   get { return age; }
   set { age = value; }
}
```

Model 与 MVC 相比，有哪些异同点？

相同点就是视图，三层架构的 UI 层相当于 MVC 中的 View 层。

不同点在于 MVC 的 Model 模型相当于三层架构中的 BLL 和 DAL，并且三层架构中没有 Controller 层，而是由单个页面上的事件处理页面与业务逻辑之间的关系。

另外 MVC 中的 View 视图和 Controller 控制器都依赖于 Model，这就体现了 Model 在 MVC 中的重要性。下面给出一段 MVC 中 Model 层代码，如下所示：

```
public class Student
{
       public string Name { get; set; }         //用户名
       public string Sex { get; set; }          //性别
       public string Age { get; set; }          //年龄
}
```

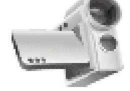
其特别之处在于，MVC 模型包含所有应用程序业务和数据访问逻辑。可以使用各种不同的技术来实现数据访问逻辑。例如，可以使用 Microsoft Entity Framework、Nhibernate、Subsonic 或 ADO.NET。

Model 模型给控制器提供了一个用户请求内容对应的数据表达(例如，用户信息)。无论我们如何向用户展示，这个数据模型都不会发生变化。这也就是我们为什么可以随意选择哪个视图来展示数据的原因。

Model 模型包含应用程序逻辑中最重要的组成部分，这些逻辑可以运用于我们需要处理的问题的过程中。

4.3　ASP.NET MVC Model 数据验证

大多数时候，我们使用前台的 JavaScript 来验证用户的输入，由于前台的 JavaScript 验证是不可靠的，所以大部分人在后台 insert 或者 update 之前做了验证，这是比较好的。另外 ASP.NET MVC 也为我们提供了很多数据验证的方法，下面就来介绍一种 Model 数据验证的方法。


视频教学：光盘/videos/04/4.3　基于 MVC 的用户登录的 Model 模型(1)　长度：8 分钟
基于 MVC 的用户登录的 Model 模型(2)　长度：10 分钟

4.3.1　实例描述

大多数动态网站都要用到信息管理功能，这就需要我们实现用户登录功能。权限认证的方式很多，但是使用账号、密码登录的认证方式以其独有的方便性一直占据主流，一时间还难以有其他方法能与其一争高下。

在这里，我们重点讲 ASP.NET MVC Model 数据验证，就不再进行数据库操作了，只是简单地模拟一下数据验证功能。

4.3.2　实例应用

【例 4-1】基于 MVC 的用户登录的 Model 模型。

前面提到，在处理数据和业务逻辑时，我们需要使用 Model。首先，我们需要对数据进行封装，还要处理简单的业务逻辑。

下面我们来创建一个 MVC 项目中的 Model 模型。首先在 Models 目录下面创建一个存放 Model 的 cs 文件，代码如下：

```
public class UserInfo
{
       /*用户信息*/
       public string LoginName { get; set; }        //登录名
       public string Password { get; set; }         //密码
       public string Username { get; set; }         //姓名
}
public class UserManager
{
       //验证登录名和密码
       public static bool Validate(string loginName, string passWord)
       {
          return "zhang" == loginName && "password" == passWord;
       }
       //根据登录名获取用户信息并返回
       public static UserInfo GetUserInfoByLoginName(string loginName)
       {
          return new UserInfo()
          {
             LoginName = "zhang",
             Password = "password",
             Username = "张三"
          };
       }
}
```

好了，这样 Model 模型就完成了，下面来创建 Controller。

我们先在 Controllers 目录下创建一个名为 AccountController 的控制器，Index 是默认的方法，我们再创建一个 Login 方法，用于接收用户登录时的提交操作。下面是 Login 方法的代码：

```
public ActionResult Login()
{
   string loginName = Request.Form["loginName"];
   string password = Request.Form["password"];
   if (Models.UserManager.Validate(loginName, password))
   {
       Session["CurrentUser"] =
Models.UserManager.GetUserByLoginName(loginName);
       return Redirect("/Account/Success");
   }
   ViewData["LoginName"] = loginName;
   ViewData["ReturnMessage"] = "用户名或者密码不正确";
   return View("Index");
}
```

这里我们首先获取用户提交的登录名和密码，然后调用相应 Model 的验证方法进行验证，如果验证成功，则记录当前用户信息，跳转到登录成功页面。

如果验证不成功，则向视图发送提示信息，返回登录视图。当然，我们得有一个 Action 用来处理成功请求。所以我们再在 Account Controller 控制器里创建一个名为 Success 的方法，方法结构参考 Index。

Controller 完成，下面就是 View 了。

我们需要一个 Index 视图作为登录页面，一个 Success 视图作为登录成功页面。

首先来看一下 Index 视图的主要代码。

```
<div>
<% Html.BeginForm("PostForm","Account"); %>
    登录名:
    <%= Html.TextBox("loginName",ViewData["LoginName"]) %><br />
    密码:
    <%= Html.Password("password") %><br />
    <input type="submit" value="Submit" />
    <%= ViewData["ReturnMessage"] %>
<% Html.EndForm(); %>
</div>
```

第四行括号中的第二个参数用来设置默认值，这里将 Controller 传递过来的数据呈现到文本框里。

这样做有什么好处呢？比如用户登录的时候可能不小心输入了不正确的密码，返回的时候，就可以自动填上用户名。

接下来我们看一下 Success 视图的主要代码。

```
<div>
<% MyMVC.Models.UserInfo currentUser =
      Session["CurrentUser"] as MyMVC.Models.UserInfo; %>
你好，欢迎你：<%= currentUser.Username %><br />
你的登录名称是：<%= currentUser.LoginName %>
</div>
```

登录成功以后将用户信息保存在 Session 中，表明当前用户已经登录。在这里可以取出用户信息，并显示出来。

当然，Session 对象存储的对象都是 object 类型的，所以取出来的时候需要对它进行转型，然后再输出。

4.3.3　运行结果

运行项目，显示结果如图 4-1 所示。

在这里，我们输入错误的密码，单击按钮后，就会提示“用户名或者密码错误”，如图 4-2 所示。

我们输入正确的用户名和密码，单击 Submit 按钮，显示结果如图 4-3 所示。

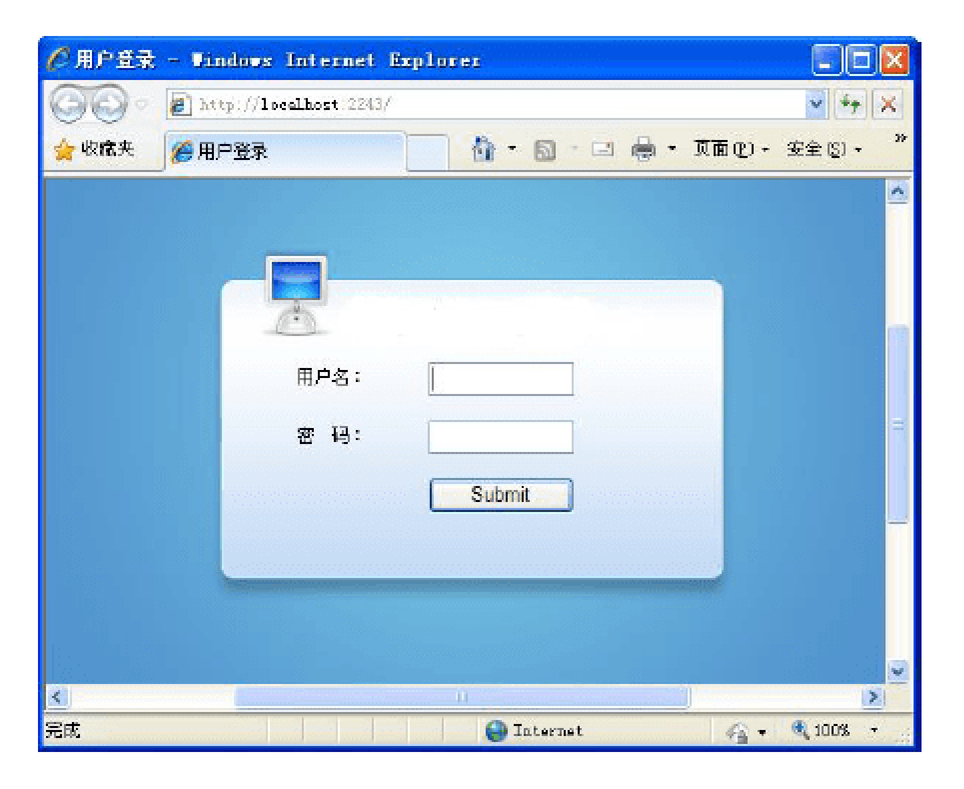



图 4-1　用户登录界面

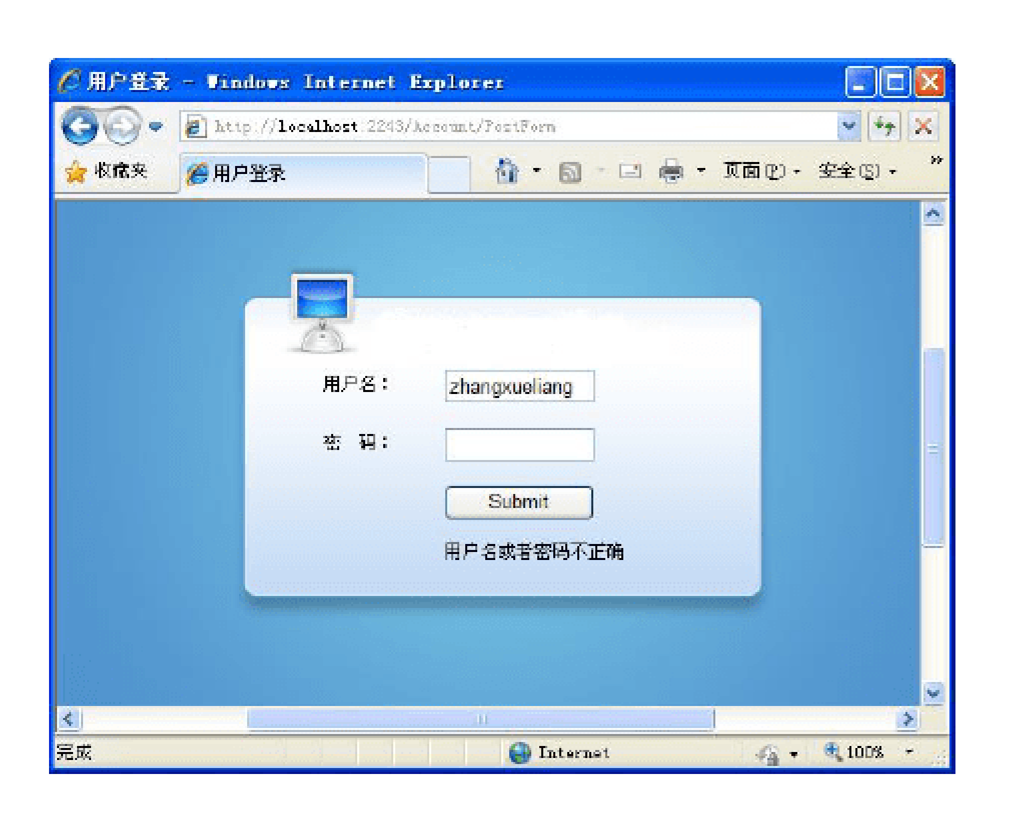


图 4-2　用户登录失败



图 4-3　用户登录成功

4.3.4　实例分析



源码解析

本实例主要了解 ASP.NET 下的 MVC 框架的用户登录问题。其中 Model 模型包含应用程序逻辑中最重要的组成部分，这些逻辑运用于我们要处理的问题的过程中。

对 Model 模型了解之后，我们创建了一个简单的 MVC 项目来了解 MVC 项目中 Model 模型的创建方式，以及 Model 模型在 MVC 中所扮演的角色。

4.4　MVC 视图模板与数据基架的结合使用

在创建 ASP.NET MVC 应用程序时，可以选择创建支持数据基架的控制器和视图。本节讲解的是如何创建一个简单的 MVC 应用程序，该应用程序将利用 MVC 支持的控制器和视图的数据模板。



视频教学：光盘/videos/04/4.4　MVC 视图模板与数据基架的结合使用　长度：20 分钟

4.4.1 基础知识

验证用户的输入在 Web 系统中经常用到，比如说输入不能为空或输入的字符要在一定的范围内等。

下面我们就来介绍一个 Model 数据验证的方法，使用的命名空间是 System.ComponentModel.DataAnnotations，在这个命名空间下，Model 为我们提供了很多数据验证的方法。

1. Required

Required 验证表达式表示的意思是必须输入，不能为空，格式如下：

```
[Required(ErrorMessage = "提示信息")]
```

例如，用户在输入的时候，用户 ID 不能为空，代码如下：

```
[Required(ErrorMessage = "ID 不能为空")]
```

2. RegularExpression

正则表达式对象用来规范一个规范的表达式(也就是说判断表达式符不符合特定的要求。比如，是不是 E-mail 的地址格式)，它具有用来检查给出的字符串是否符合规定的属性和方法，其格式如下所示：

```
[RegularExpression("正则表达式",ErrorMessage = "提示信息")]
```

例如，只允许输入“男”或“女”，代码如下：

```
[RegularExpression("^[\u7537\u5973]+$",ErrorMessage = "性别只能输入“男”或“女”")]
```

3. Range

Range 验证表达式用于创建并返回一个包含指定范围的元素的数组，用于验证用户输入是否在指定范围内，其格式如下：

```
Range(min,max,ErrorMessage="提示信息")
```

例如，允许输入的年龄在 1～120 岁之间，代码如下：

```
[Range(1, 120, ErrorMessage = "年龄必须在 1～120 岁之间")]
```

4.4.2 实例描述

在本实例中，将会添加一个已经包含用于显示、编辑和更新模型数据的操作方法的控制器(Controller)。通过使用 ASP.NET MVC 内置的数据基架，将生成基于所定义模型的视图，下面来具体讲解。

4.4.3 实例应用

【例 4-2】MVC 视图模板与数据基架的结合使用。

首先，我们需要创建新的 MVC 项目。在这里，我们选择 ASP.NET MVC 2 Web 应用程序，修改项目名称为 MvcDataViews。在创建测试项目对话框中，我们选中【否，不创建单元测试项目】。

下面创建模型类 Model。我们使用一个简单的数据模型，实现视图模板添加、编辑和显示该模型的值。我们来创建一个学生类，其中包含 4 个字段，分别是：学生编号、学生姓名、性别以及年龄。

在“解决方案资源管理器”中，右击 Models 文件夹，再选择【添加】命令，然后选择“类”，并修改名字为 Student。具体代码如下：

```
public class Student
{
        [Required(ErrorMessage = "ID 不能为空")]
        public int Id { get; set; }
        [Required(ErrorMessage = "姓名不能为空")]
        public string Name { get; set; }
        [RegularExpression("^[\u7537\u5973]+$",ErrorMessage = "性别只能输入“男”
或“女”")]
        public string Sex { get; set; }
        [Range(1, 120, ErrorMessage = "年龄必须在 1～120 岁之间")]
        public int Age { get; set; }
}
```

在完成 Model 代码后，右击并选择【生成】。



此为必须步骤，因为 Visual Studio 在根据 MVC 模板生成控制器代码和视图标记时，将使用该模型实例。

Model 完成后，再来创建控制器。

在【解决方案资源管理器】中，右击 Controller 文件夹，再选择【添加】命令，然后选择“控制器”，并修改名字为 StudentController。选中【为“创建”、“更新”和“详细信息”方案添加操作方法】复选框。

该控制器应该包含的操作方法有：Index、Details、Create(用于 HTTP GET 及 HTTP POST)、Delete(用于 HTTP GET 及 HTTP POST)和 Edit(用于 HTTP GET 及 HTTP POST)。

在 StudentController 类顶部添加下面一段代码：

```
static List<Student> students = new List<Student>();
```

此代码用于创建 Student 对象的列表。

接下来该添加数据视图了。

1. Index 视图

首先，我们来创建 Index 视图，它是 Index 方法呈现的列表视图，此视图用来显示我们创建的 Student 对象。另外，每行还包括用于在 Details 视图和 Edit 视图中显示的链接。

下面来添加列表视图。

(1) 首先找到 StudentController 类里的 Index 方法，然后在方法名上右击，选择【添加视图】命令。

(2) 在【视图名称】文本框中输入 Index。
(3) 选中【创建强类型视图】复选框。
(4) 从【视图数据类】列表中选择 MvcDataViews.Models.Student。
(5) 从【视图内容】列表中选择 List，同时选中【选择母版页】复选框。
(6) 单击【添加】按钮，将会在新建的 Student 文件夹内创建 Index 视图；Index 视图包含用于显示数据列表的 MVC 模板。

在 Index 视图中，需对 Html.ActionLink 控件进行如下修改：

```
<%: Html.ActionLink("Edit", "Edit", new { id=item.Id}) %>
<%: Html.ActionLink("Details", "Details", item)%>
<%: Html.ActionLink("Delete", "Delete", new {  id=item.Id })%>
```

还需要用以下代码替换 StudentController 中的 Index 方法。

```
    public ActionResult Index()
    {
        return View(students);
    }
```

2. Create 视图

Create 是一个用于创建 Student 对象的视图。在创建 Student 对象时，需要定义该学生的 ID、名字、性别和年龄。



StudentController 有两个 Create 操作方法。一个用于接收 HTTP GET 请求并呈现 Create 视图，另一个从 Create 视图接收 HTTP POST 请求，检查数据有效性，向列表添加数据以及重定向到 Index 操作方法。

下面来创建 Create 视图。
(1) 首先在 StudentController 类中找到处理 HTTP GET 的 Create 方法。
(2) 在 Create 操作方法内右击，选择【添加视图】命令。
(3) 在【视图名称】文本框中输入 Create。
(4) 选中【创建强类型视图】复选框。
(5) 从【视图数据类】列表中选择 MvcDataViews.Models.Student。
(6) 从【视图内容】列表中选择 Create。
(7) 选中【选择母版页】复选框，单击【添加】按钮，向项目中添加一个名为 Create 的视图。

另外，还要在 StudentController 类中找到 HTTP POST 的 Create 操作方法，并用以下代码进行替换。

```
    [HttpPost]
    public ActionResult Create(Student s)
    {
        try
        {
            if (!ModelState.IsValid)
            {
                return View("Create", s);
            }
```

```
        students.Add(s);                //把数据添加到 student 中
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

3. Details 视图

Details 视图用于显示单一 Student 对象的信息。此视图提供到 Edit 视图的链接以及用于返回列表视图的链接。

下面来创建 Details 视图。

(1) 首先在 StudentController 类中找到 Details 操作方法。

(2) 在操作方法内右击，选择【添加视图】命令。

(3) 在【视图名称】文本框中输入 Details。

(4) 选中【创建强类型视图】复选框。

(5) 从【试图数据类】列表中选择 MvcDataViews.Models.Student。

(6) 从【视图内容】列表中选择 Details。

(7) 选中【选择母版页】复选框，单击【添加】按钮，就添加一个名为 Details 的视图。

另外，需要找到 Details 视图的 Html.ActionLink 控件，并对其代码进行修改，修改后的代码如下：

```
<%: Html.ActionLink("Edit", "Edit", new { id=Model.Id }) %> |
<%: Html.ActionLink("Back to List", "Index") %>
```

用以下代码替换 StudentController 中的 Details 方法。

```
public ActionResult Details(Student s)
{
    return View(s);
}
```

4. Delete 视图

通过使用 Delete 视图，用户可以从列表中删除 Student 对象。可以选择删除 Student 对象，也可以返回列表视图。



StudentController 有两个 Delete 操作方法。一个用于接收 HTTP GET 请求并呈现 Delete 视图，另一个从 Delete 视图接收 HTTP POST 请求，移除所选的对象以及重定向到 Index 操作方法。

下面来创建 Delete 视图。

(1) 首先在 StudentController 类中找到处理 HTTP GET 的 Delete 方法。

(2) 在 Delete 操作方法内右击，选择【添加视图】命令。

(3) 在【视图名称】文本框中输入 Delete。

(4) 选中【创建强类型视图】复选框。

(5) 从【试图数据类】列表中选择 MvcDataViews.Models.Student。

(6) 从【视图内容】列表中选择 Delete。

(7) 选中【选择母版页】复选框，单击【添加】按钮，添加一个名为 Delete 的视图。

还需要在 StudentController 类中找到处理 HTTP GET 的 Delete 操作方法，并将其代码修改如下：

```
public ActionResult Delete(int id)
{
    Student s = new Student();
    foreach (Student stu in students)
    {
        if (stu.Id == id)
        {
            s.Id = stu.Id;
            s.Name = stu.Name;
            s.Sex = stu.Sex;
            s.Age = stu.Age;
        }
    }
    return View(s);
}
```

把处理 HTTP POST 的 Delete 操作方法修改如下：

```
[HttpPost]
public ActionResult Delete(Student s)
{
    try
    {
        foreach (Student stu in students)
        {
            if (stu.Id == s.Id)
            {
                students.Remove(stu);
                break;                    //移除以后跳出循环
            }
        }
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

5. Edit 视图

通过使用 Edit 视图，可以为 Student 对象更改值。



StudentController 有两个 Edit 操作方法。一个用于接收 HTTP GET 请求并呈现 Edit 视图，另一个从 Edit 视图接收 HTTP POST 请求，检查数据有效性，更新对应的 Student 中的数据以及重定向到 Index 操作方法。

下面我们来创建 Edit 视图。

(1) 首先在 StudentController 类中找到处理 HTTP GET 的 Edit 方法。

(2) 在 Edit 操作方法内右击，选择【添加视图】命令。
(3) 在【视图名称】文本框中输入 Edit。
(4) 选中【创建强类型视图】复选框。
(5) 从【试图数据类】列表中选择 MvcDataViews.Models.Student。
(6) 从【视图内容】列表中选择 Edit。
(7) 选中【选择母版页】复选框，单击【添加】按钮，添加一个名为 Edit 的视图。
还需要修改 StudentController 类中处理 HTTP GET 的 Edit 操作方法，其代码如下：

```
public ActionResult Edit(int id)
{
    Student s = new Student();
    foreach (Student stu in students)
    {
        if (stu.Id == id)
        {
            //根据所选的 id 把值赋给 Student 对象 s
            s.Id = stu.Id;
            s.Name = stu.Name;
            s.Sex = stu.Sex;
            s.Age = stu.Age;
        }
    }
    return View(s);
}
```

此外，还需用以下代码替换 HTTP POST 的 Edit 操作方法。

```
[HttpPost]
public ActionResult Edit(Student s)
{
    if (!ModelState.IsValid)
    {
        return View("Edit", s);
    }
    foreach (Student stu in students)
    {
        if (stu.Id == s.Id)
        {
            //根据 id 把 Student 对象 s 的信息赋给泛型 stu
            stu.Id = s.Id;
            stu.Name = s.Name;
            stu.Sex = s.Sex;
            stu.Age = s.Age;
        }
    }
    return RedirectToAction("Index");
}
```

最后，我们还应将主页上的链接添加到个人列表。首先，找到 Home 文件夹，打开 Index 视图，然后添加下面一段代码。

```
<%: Html.ActionLink("学生列表","Index","Student") %>
```

完成之后就可以编译运行了。

4.4.4　运行结果

运行项目，单击“学生列表”超链接，显示结果如图 4-4 所示。

单击 Create New，进入 Create 页面，由于在 Model 中对输入信息进行了限制，所以如果所输入的信息不符合要求，那么在文本框后面会给出提示信息。另外，单击下面的 Back to List，可以返回学生列表，如图 4-5 所示。





图 4-4　Index 页面



图 4-5　Create 页面

单击对应的 Edit，此时会显示对应的 Edit 视图，可对所选信息进行修改，然后单击 Save 按钮，此时会重新显示 Index 视图，其中更改的数据已经更新。

另外，在我们单击下方的 Back to List 以后，会跳转到学生列表页面(Index 视图)，效果如图 4-6 所示。

单击对应的 Details，可查看所选学生的详细信息。同时单击下方的 Edit 后，还可以跳转到 Edit 页面，对所选学生的信息进行修改。还有 Back to List，单击之后可以跳转到学生列表页面，效果如图 4-7 所示。



图 4-6　Edit 页面



图 4-7　Details 页面

单击对应的 Delete 按钮，可以跳转到 Delete 页面。单击 Delete 按钮则删除该信息，同样可以返回学生列表页面，效果如图 4-8 所示。



图 4-8　Delete 页面

4.4.5　实例分析



源码解析

本实例主要学习 ASP.NET 下的 MVC 视图模板与数据基架的结合使用。示例中添加了一个已经包含用于显示、编辑和更新模型数据的操作方法的控制器。

通过使用 ASP.NET MVC 内置的数据基架，还生成了基于所定义模型的视图。另外，在该应用程序中，我们利用了 MVC 支持的控制器和视图的数据模板。

4.5　常见问题解答

4.5.1　ASP.NET MVC 中的 M、V 和 C 可以各自独立开发吗



关于 ASP.NET MVC 中的 M、V 和 C 真的可以各自独立开发吗？是否有所限制？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

关于 ASP.NET MVC 中的 M、V 和 C 真的可以各自独立开发吗？是否会有所限制？

【解决办法】可以，但没那么绝对！完全独立开发虽然可行，但是如果你真的这么做就会绊手绊脚的，而且缺乏工具支援。我指的“完全独立开发”是指一开始就让 M、V 和 C 独立开发，没什么必要，但是可行。

M、V 和 C 的关系是既分工又合作，必须在有点黏又不能太黏的情况下，才能发挥 ASP.NET MVC 的优势。

依照我的开发经验，我觉得 M(Model)是 MVC 架构的中心。有了 Model 之后，就可以让 Controller 与 View 参考这些 Model(模型)，并且进行分工开发，最后再进行整合，这是我认为

最有效的开发方法。

4.5.2 MVC 架构中的模型部分做什么用



MVC 架构中的模型部分做什么用？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

MVC 架构中的模型部分做什么用？

【解决办法】模型表示企业数据和业务规则。在 MVC 三个部件中，模型拥有最多的处理任务。模型能为视图提供数据，并且模型的代码只需写一次就可被多个视图重用，所以减少了代码的重复性。

因为模型是自包含的，并且与视图和控制器分离，所以很容易改变应用程序的数据层和业务规则。如果你想把数据库从 MySQL 移植到 Oracle，只需改变你的模型即可。一旦你正确地实现了模型，不管你的数据来自数据库还是 LDAP 服务器，视图都会正确显示。

模型也有状态管理和数据持久性处理的功能。例如，基于会话的购物车和电子商务过程也能被 Flash 网站或者无线联网的应用程序所重用。

4.6 习 题

一、填空题

(1) 业务模型还有一个非常重要的模型那就是__________。

(2) __________的设计可以说是 MVC 的核心。

(3) MVC 并没有提供模型的设计方法，只告诉你应该组织管理这些模型，以便于__________和__________。

(4) 数据模型主要指__________。

(5) 在 MVC 的 3 个部件中，__________拥有最多的处理任务。

二、选择题

(1) 模型不包含应用问题的__________，它封装了所需的数据，提供了完成问题处理的操作过程。

A. 核心数据　　B. 逻辑运算
C. 逻辑关系　　D. 计算功能

(2) 在基于窗口的 GUI 应用程序中，__________一般就是控件的事件处理函数。

A. Model　　B. View
C. Controller　　D. DAL

(3) 一个应用的业务流程或者业务规则的改变只需改动 MVC 的__________。

A. View　　B. BLL
C. Controller　　D. Model

三、上机练习

上机练习： 自定义 Model 模型的验证。

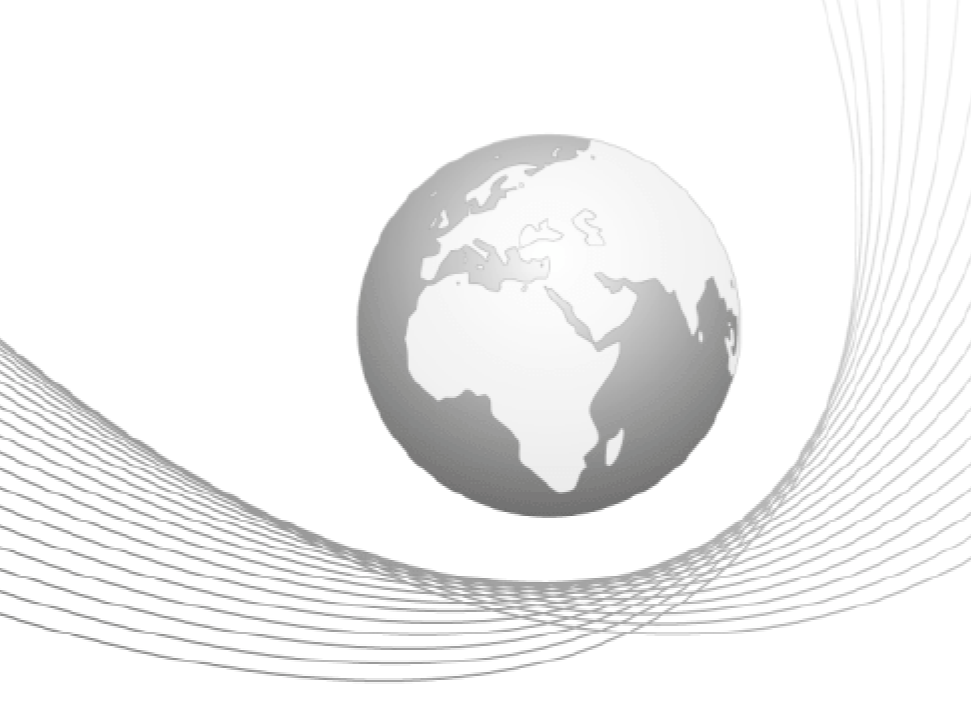
自定义一个 Model 模型的验证，字段包括用户名(UserName)、密码(PassWord)、邮箱(E-mail)和年龄(Age)。要求如下：

(1) 所有字段不能为空。

(2) 密码长度大于 6 位小于 12 位。

(3) 邮箱地址要包含@和。

(4) 年龄在 1～120 岁。



第 5 章　简单实现绚丽的界面

内容摘要

用户对一个软件产品的评价都是从视图(View)开始的。虽然我们可以很好地设计程序逻辑，很好地分离控制器，但是对于应用程序的用户来说是不可见的，用户很难直观地感受到程序架构逻辑设计的优秀。

对用户而言，应用程序的人性化、美观和易用都是从视图中体现出来的。对应用程序的最终使用者来说，视图就是应用程序，视图的优秀程度基本上代表了应用程序产品的质量。

换言之，如果在你的应用程序中的程序逻辑有些小小的不完美，一般不会很直观地引起用户的抵制情绪，但是如果该应用程序用户界面做得很丑、很痛苦，那么你的应用程序基本上可以说是报废了。

本章不是讲如何让你做出一个漂亮的页面，因为那是 HTML 和 CSS 的任务。本章主要讲如何让开发人员更好、很方便地组织 ASP.NET MVC 页面中的代码，包括如何在 Controller 向 View 中传递数据。

学习目标

- 了解什么是 View 以及 View 在 MVC 中的地位
- 掌握 Controller 向页面视图传递数据的方法
- 掌握强类型的数据传递方式

5.1　ASP.NET MVC 中的 V

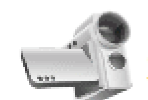
前面讲过，ASP.NET MVC 的核心是 Controller，它是整个应用程序的调配者和管理者。

前面我们使用了没有视图的 Controller 来响应用户请求，但是看着 Controller 中那些大段的 Response.WriteLine()方法，是不是觉得非常崩溃呢？假如我们的页面代码非常多——可能有数千行 HTML 代码，那么可以想象这个 Controller 的模样会多么让人发狂。

而且这种方式只能将页面代码以字符串的形式组织，进行所见即所得的编辑更是不可能的事情。我们只能在 Dreamweaver 中编辑完以后 Copy 到 Controller 里进行输出。

维护——将会是另一件更加痛苦的事情。

有了 View(视图)，这些问题就迎刃而解了。


视频教学：光盘/videos/05/5.1　ASP.NET MVC 中的 V　　长度：10 分钟

5.1.1　基础知识

View 称为视图。在 MVC 架构的应用程序中 View 承担着组织用户界面的任务。它能够操作 Model，并把 Model 中的数据以适当的格式合理地组织到页面当中，提供给用户。

1. 为什么要用 View

虽然前面我们已经证明过 View 并不是必要的，但是假如真的没有 View，MVC 就会很快没有用处的。

例如我们要在访问 Home/Index 的时候向页面输出一行“Hello World！”，我们的 HomeController 下面的 Index 方法就需要这样来编写：

```
public void Index()
{
    string msg = "Hello World!";
    Response.Write("<html>\n");
    Response.Write("<head>\n");
    Response.Write("<title>SayHello</title>\n");
    Response.Write("</head>\n");
    Response.Write("<body>\n");
    Response.Write("<h1>");
    Response.Write(msg);
    Response.Write("</h1>\n");
    Response.Write("</body>\n");
    Response.Write("</html>\n");
}
```

假如我们要向用户打印一个类似网易、新浪之类的门户首页，想必每个人都会选择自杀的，因为我刚查看了网易的首页有 4 700 多行 HTML 代码，新浪首页有 7 300 多行代码。完成这样一个门户系统，工作量可以说就像让你一个人以纯手工的方式去建设三峡大坝。

在 ASP.NET MVC 中使用 View，就可以像传统的 WebForm 一样，拖放一些 HTML 控件

到 aspx 页面或 ascx 用户控件中。

在 Visual Studio 里还可以使用纯 HTML 方式或者可视化的方式对页面中的 HTML 元素进行操作，如图 5-1 所示。

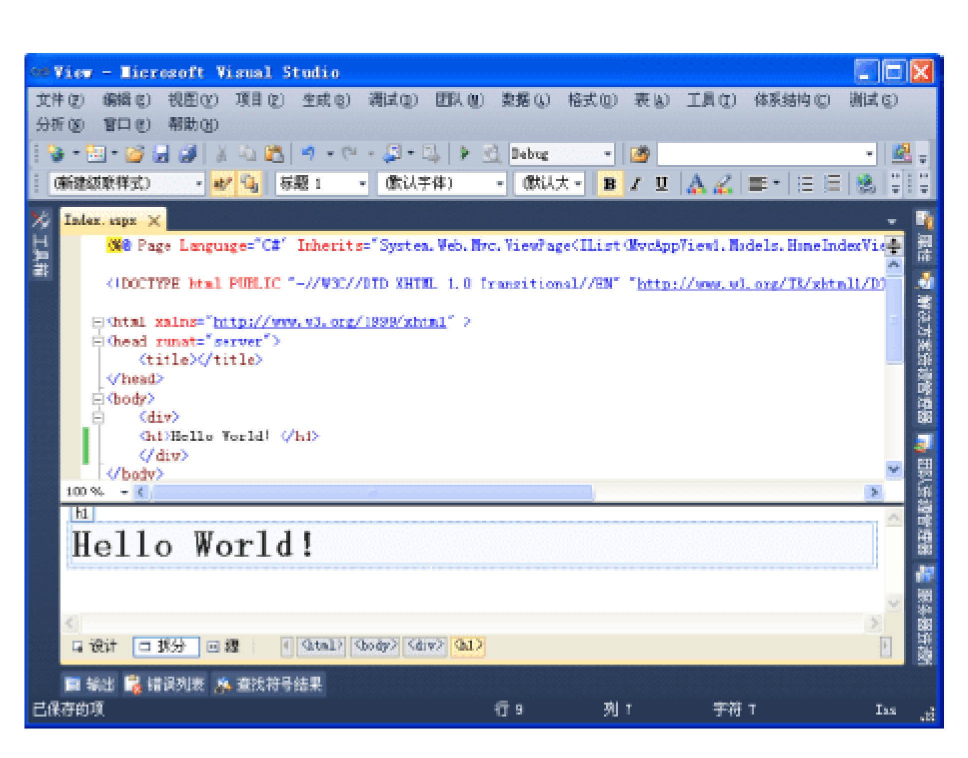


图 5-1　Visual Studio 2010 操作界面

将编写好的视图文件保存到站点根目录下的 Views 目录中相应的控制器对应的目录里，然后，就可以在 Controller 中使用这些视图了。

2．如何在 Controller 中使用 View

在前面的章节中我们讲过，Controller 里的 Action 可以使用 View()之类的方法返回一个 ActionResult 类型的对象，代码如下：

```
public ActionResult Index()
{
    return View();
}
```

在执行完该 Action 以后，系统接收到 ActionResult 类型的对象，会自动根据相应的值调用相应的 View 来处理用户的请求。

那么，在上面的代码中，我们并没有指定视图文件的地址，系统如何寻找视图文件呢？

例如上面的代码是位于 HomeController 下的 Index 方法，在 Views 目录下的 Home 目录里没有任何文件，程序会是什么样的执行结果呢？

运行一下，结果如图 5-2 所示。

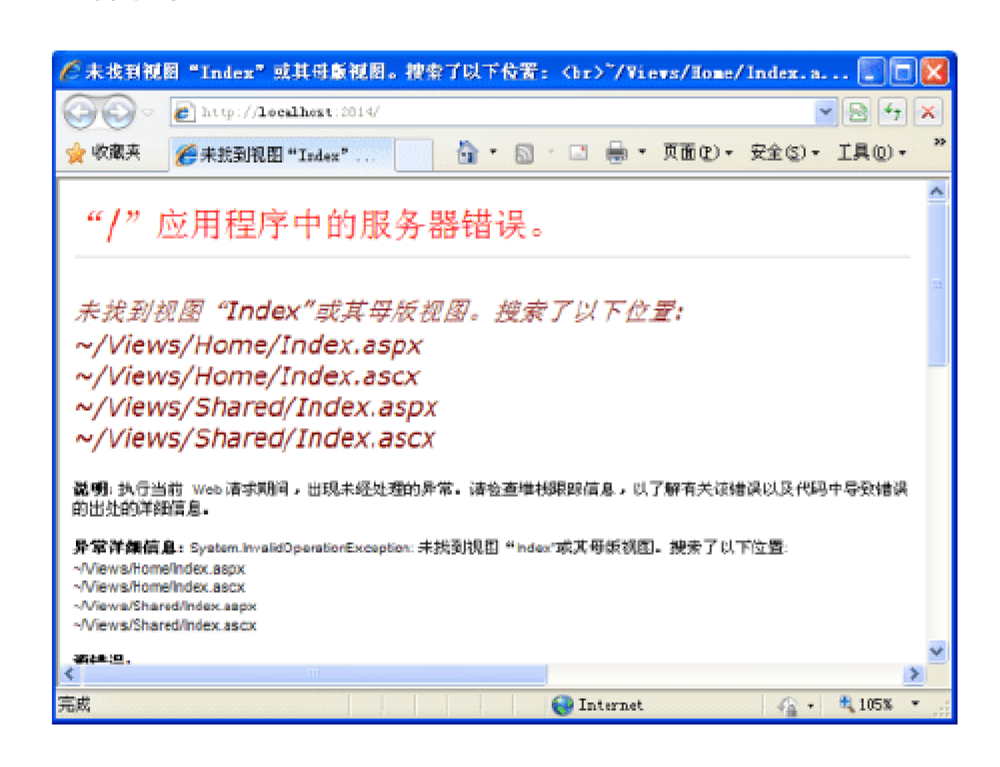


图 5-2　没有视图文件时的执行结果

出错了！理所当然。

不过就在这个出错了的页面中，我们得到了一个非常重要的信息：ASP.NET MVC 对视图的管理使用的是自动搜索的管理机制，编译期并不检查视图是否存在(其实检查了也没什么意义)，只在程序执行的时候才去 Views 目录下查找对应 Controller 名称的子目录，然后在子目录中查找与所请求的 Action 名称相同的视图文件名称。

ASP.NET MVC 接收两种视图文件格式：aspx(ASP.NET 页面)和 ascx(ASP.NET 用户控件)。

在 Views 目录下面还有一个 Shared(共享)目录，存放一些整个应用程序中各 Controller 之间共享的视图文件，所以系统在 Home 目录中没有找到对应的视图文件的时候，自动搜索了 Shared 目录，如果 Shared 目录中有用户要调用的视图文件，系统也会正常调用。

3. ASP.NET MVC 的页面模型

从前面的学习中大家应该感觉到：ASP.NET MVC 默认使用的视图引擎是 ASP.NET WebForm 中的页面模型(Web 页面 aspx 和用户控件 ascx)。

在 ASP.NET MVC 的页面模型中主要包含以下几个部分。

- master　母版页，ASP.NET 页面模板框架。用于搭建风格和页面主体结构相同的多个页面的布局框架。
- ascx　用户控件，ASP.NET 页面模块。用于拼组不同页面中结构相同的页面视图模块。在 ASP.NET MVC 中，ascx 被视为局部视图，可以被系统调用以处理用户请求。
- aspx　ASP.NET 页面内容的主体，在该部分中可以直接引入母版页或用户控件。在 ASP.NET WebForm 中，可以使用浏览器直接访问这些资源文件。

在 ASP.NET WebForm 中，页面模型类所在的命名空间为 System.Web.UI，而 ASP.NET MVC 的页面模型类所在的命名空间为 System.Web.Mvc。

以 ASP.NET 页面类为例，在 ASP.NET WebForm 中，它继承自 System.Web.UI.Page 类，而在 ASP.NET MVC 中，它继承自 System.Web.Mvc.ViewPage 类。也就是说，如果在 ASP.NET MVC 应用程序中，一个 aspx 页面要被作为 ASP.NET MVC 的默认视图来使用，那么这个页面必须继承自 System.Web.Mvc.ViewPage 类。

同理，母版页 master 和用户控件 ascs 也是一样。

4. View 要做的事情

想必大多数使用过 ASP.NET WebForm 的人第一次看到 ASP.NET MVC 的时候，都会有些失望地叹息：又回到上个世纪末了！

确实，上个世纪末流行的 ASP 语言一直面临着的问题随着 ASP.NET WebForm 的出现骤然消失，却又随着 ASP.NET MVC 的诞生又被摆到我们的桌面上来。那就是所谓的“炸酱面”式的代码。



这里所描述的也就是外国人喜欢的“意大利面条”式的代码，外文书籍中通常是这种说法，都是一个意思。

在 ASP 之类的上一代开发语言中，通常会在页面中大量使用<% %>语法去隔离服务器端代码和页面 HTML 内容。在处理一些细节的业务的时候，页面中往往会有大量的<% %>混乱地组合在一起，很像炸酱面里的肉末，根本无法控制、掺和在一根根面条中间。那些代码，着

实让人看着很纠结，很纠结……

不过，ASP.NET MVC 还处于一个初期发展的阶段，相信这个问题会很快被解决掉。

1) View 的职责

View 的职责是向客户端提供用户界面。它可以引用模型(Model)，将模型中的数据格式化后展示给用户。在 ASP.NET MVC 中，包括接收和检查控制器通过 ViewData 字典(通过 ViewData 属性访问)传递过来的数据，以及将其格式化为 HTML。

由于 MVC 天生的特性决定了代码的分离，数据、业务的封装都独立在一些类里，View 里仅仅处理页面显示内容，相对于早期的“炸酱面”式的代码，这里清晰了许多。

就像一碗炸酱面，只放了几滴酱汤，放眼过去，碗里基本上都是面条，就不会再有十分纠结的感觉了。

2) View 的禁忌

回想 ASP 程序的代码，并不是说使用<% %>语法去隔离开的程序就很纠结，而是因为 ASP 技术的局限性，使页面中数据访问、IO 操作、业务逻辑和页面展示等各个功能的代码都杂乱地摆放在一起，导致整个页面功能太杂，以至于编写和维护都很痛苦。

在 ASP.NET MVC 中，架构的模式已经决定了功能的分离，所有的业务逻辑之类的代码全部放在 Model 中处理，所有展示逻辑都使用 Controller 统一调配，剩下的 View 只需专心致志地执行数据展示的功能即可。

反过来说，将一切处理应用程序逻辑(包括视图逻辑和业务逻辑)的代码放入 View 中都是不可谅解的，甚至可以说是不可饶恕的。

除非你想重新编写“炸酱面”式的代码。

5.1.2　实例描述

ASP.NET MVC 是微软公司推出的一种框架模型。多少年来，微软一直坚持的理念就是：做傻瓜式的产品。虽然前面这么大篇幅讲的东西，实际在使用过程中十分简单。

接下来的这个实例作为 View 入门实例，我们将简单地模拟一个加法计算器来演示 View 的用法。

5.1.3　实例应用

【例 5-1】ASP.NET MVC 中的 V。

(1) 首先创建一个“ASP.NET MVC 2 空 Web 应用程序”。创建完以后程序只搭好了架构，添加了一个默认的路由规则，并没有创建任何的 Controller、Model 或 View。

(2) 接下来创建一个 Controller，命名为 HomeController。创建完以后系统自动添加一个名为 Index 的 Action。



因为在应用程序默认配置的路由规则中，Controller 参数的默认值为 Home，Action 的默认值为 Index，所以这里创建一个带 Index 动作的 HomeController 就可以直接请求到该动作了。

(3) 创建 HomeController 下 Index 动作对应的视图文件。创建该文件以前我们先来创建一个母版页，保存在 Views/Shared 目录下，并添加页面结构代码。接着在 Views 目录下创建一个 Home 目录，并在 Home 目录中创建一个普通视图，命名为 Index.aspx，注意该页面套用刚才创建的母版页。

(4) 修改 Index.aspx 视图代码，添加一个执行加法运算的表单，页面完整代码如下：

```
<%@ Page Title="" Language="C#"
MasterPageFile="~/Views/Shared/MyViewMaster.Master"
Inherits="System.Web.Mvc.ViewPage<dynamic>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent"
runat="server">
Index
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <form action="/Home/Add" method="post">
    <input name="value1" /><br />
    +
    <input name="value2" /><br />
    <hr />
    <button type="submit">相加</button>
</form>
</asp:Content>
```

(5) 在上面的代码表单中，提交目标是/Home/Add，说明 HomeController 中还需要一个名为 Add 的 Action，用来处理加法操作。这里我们为其添加一个名为 Add 的 Action，代码如下：

```
public ActionResult Add()
{
    int value1 = int.Parse(Request.Form["value1"]);
    int value2 = int.Parse(Request.Form["value2"]);

    int result = value1 + value2;
    ViewData["Result"] = result;

    return View();
}
```

(6) 相应的，我们还要在 Home 目录下添加一个名为 Add 的视图文件。这里添加一个 Add.aspx 文件，页面也继承自刚才创建的母版页，页面主要部分代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
<h3>
    加法的执行结果为：
    <%= ViewData["Result"] %>
</h3>
<a href="/">返回</a>
</asp:Content>
```

5.1.4 运行结果

运行该应用程序，结果如图 5-3 所示。

在上下两个文本框里分别输入数字，然后单击【相加】按扭，执行加法运算后的页面如图 5-4 所示。

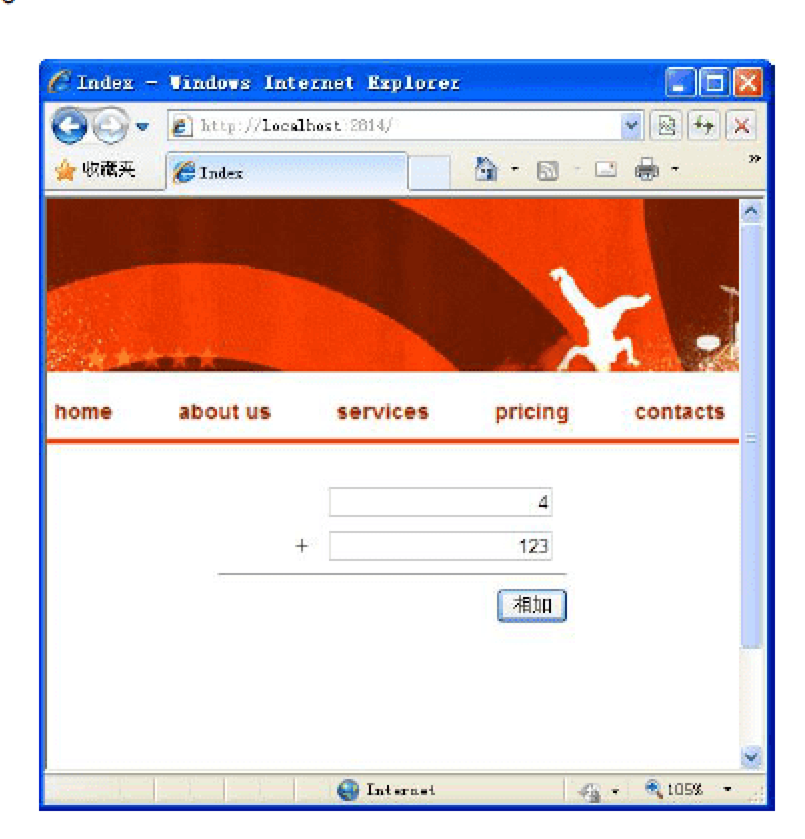


图 5-3　加法计算器

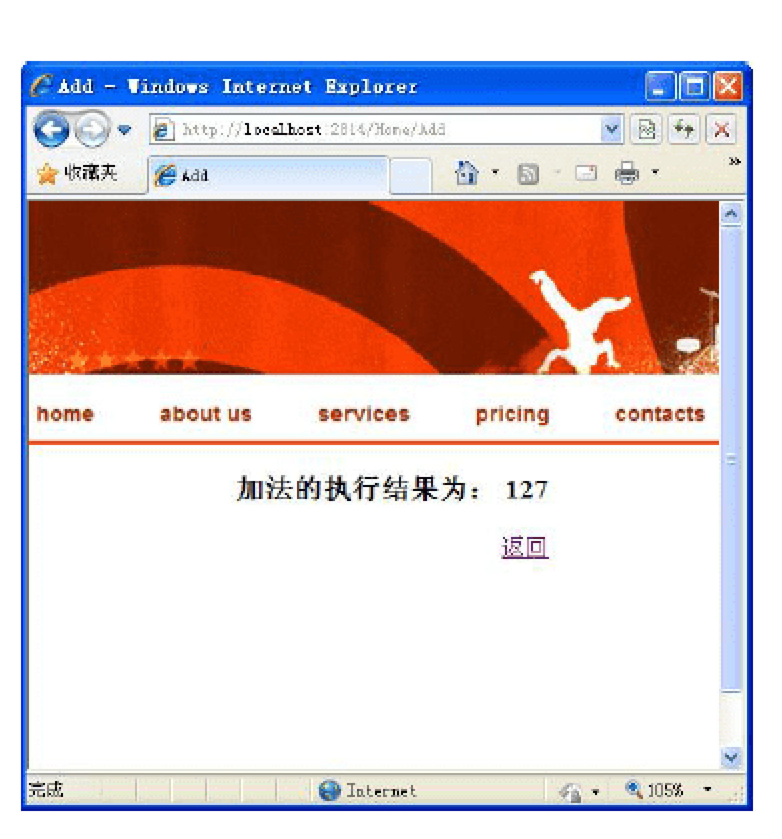


图 5-4　计算结果

5.1.5　实例分析



源码解析

该实例共创建了 3 个 View(视图)文件，其中一个是母版页，另外两个是普通的 aspx 页面 Index.aspx 和 Add.aspx。

其中 Index.aspx 页面响应动作触发，在页面上显示一个执行加法运算的表单。注意这个表单中使用的都是 HTML 标签，没有使用 WebForm 常用的服务器控件。Add.aspx 页面使用 ViewData 字典接收 Controller 传递过来的数据，直接打印到页面中。关于 ViewData 的用法将在下一节中详细讲解。

5.2　实现用户注册确认页面

从前面讲解的 MVC 程序的运行机制可以得知：为了降低耦合，MVC 的 3 个模块之间的关系十分独立。

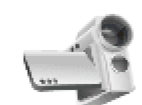
对于用户界面来说，Controller 只负责数据的存取和界面逻辑的实现；View 只负责界面的呈现，并且在界面的呈现过程中 View 不能进行数据的存取和逻辑处理(虽然它可以，但是不建议这么做)，数据的存取和逻辑处理都要交给 Controller 负责。

那么当 Controller 在处理完数据以后，如何把数据交给 View 进行展示呢？

对于这个问题，我们可以使用 Session 对象进行暂存。但是 Session 对象在整个会话生存期内都有效，如果用户操作较多，会话时间稍长，那么对服务器的内存消耗相当大。要解决这个问题，就要编写繁琐的代码进行 Session 数据管理，非常费时费力。

ASP.NET MVC 在这个问题上提供了一个很好的解决方案，那就是使用 ViewData 和 TempData。

其实在前面章节中我们已经简单使用过 ViewData，只是没有做详细的解释，下面我们来研究一下它们。

视频教学：光盘/videos/05/5.2　实现用户注册确认页面　　长度：12 分钟

5.2.1　基础知识

ASP.NET MVC 提供了两个数据字典 ViewData 和 TempData，它们可以将 Controller 处理过的数据临时存储起来，在 View 中取出来呈现给用户。

例如，正如前面已经使用过的方法，我们可以在 Controller 里面使用 ViewData 保存一些数据：

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        string msg = "Hello World!";
        ViewData["msg"] = msg;

        return View();
    }
}
```

接下来可以在对应的 View 中直接取出该数据，再使用表达式显示在页面中：

```
<h1>
<%= ViewData["msg"] %>
</h1>
```



当然，这里还可以保存其他类型的对象。只不过在取出数据的时候需要进行类型转换。

该语法看起来很熟悉，对吗？不错。这是一个字典(Dictionary)，其用法和一个普通的 Dictionary 对象一样。

TempData 对象的使用方式和语法与 ViewData 对象完全一样，只是其使用条件有稍许不同。这点稍后讲解，现在来了解一下这两个对象。

1. ViewData 和 TempData 对象

ViewData 对象声明在 IViewDataContainer 接口(命名空间 System.Web.Mvc)中。

```
public ViewDataDictionary ViewData { get; set; }
```

后来被 ViewPage 类(命名空间为 System.Web.Mvc)实现 IViewDataContainer 接口的时候对其进行了实现。

再后来 ViewPage 类被 ViewPage<TModel>(命名空间为 System.Web.Mvc)类继承的时候又进行了一次重写，重写的声明代码如下：

```
public ViewDataDictionary<TModel> ViewData { get; set; }
```

TempData 对象声明在 ViewPage 类中，其声明结构如下：

```
public TempDataDictionary TempData { get; }
```

ViewData 和 TempData 对象所属的类 TempDataDictionary 和 ViewDataDictionary(命名空间均为 System.Web.Mvc) 都实现了 IDictionary<string, object> 接口(命名空间为 System.Collections.Generic)，所以它们具备字典对象的基本特征。

2. ViewData 和 TempData 对象的区别

既然二者的用法基本一样，那么它们有什么区别呢？

首先从字面上理解：ViewData 对象是一个视图数据字典，TempData 对象是一个临时数据字典。

ViewData 对象前面已经使用过，它能够在 Controller 中保存一些数据，在与之关联的 View 中读取出来，所以 ViewData 对象也可以称为数据呈现字典。不过有一点局限性：ViewData 对象只能在当次请求时被访问。

而 TempData 对象也可以在 Controller 中保存一些数据，但是它的生存周期可不只是当前请求，它可以在该会话的任何一次请求中被访问。不过它在某一次请求访问过以后，就随该请求自动销毁，所以 TempData 对象被称为临时数据呈现字典，也有人说是跨页数据呈现字典，这些叫法都可以。

也就是说，ViewData 和 TempData 对象的区别在于 ViewData 对象的生存周期是当次请求；而 TempData 对象的生存周期是当前会话，不过只能被一次性使用。

例如用户在一次会话中分别依次请求了/Home/Index、/Home/Show 和/Home/List 3 个(或更多)URL，如图 5-5 所示。

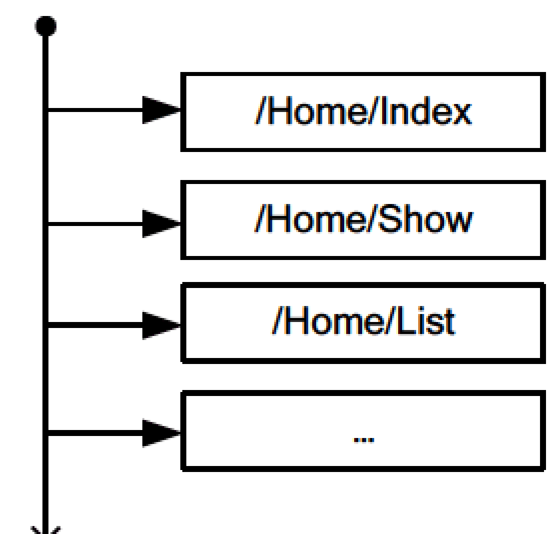


图 5-5　某次会话的请求流程

假设在执行/Home/Index 请求的时候，我们分别在 ViewData 和 TempData 对象中保存了一个数据。那么，ViewData 中保存的数据只能在/Home/Index 请求的当次被取出并呈现，而 TempData 中保存的数据可以在/Home/Index、/Home/Show、/Home/List 或以后的其他某次请求中使用。

例如我们在执行/Home/Index 请求的时候取出了 TempData 对象中保存的数据，那么在执行/Home/Show 以及以后的请求中将不存在这些数据。如果在执行/Home/Show 请求的时候取出了 TempData 中保存的数据，那么在执行/Home/List 以及以后的请求的时候将无法得到所保存的数据。

也可以简单地理解为：ViewData 将数据保存到 Request 中，而 TempData 将数据保存到 Session 中，只不过这个 Session 是个一次性的 Session 而已。

5.2.2　实例描述

在应用程序中，注册一些信息通常需要用户提交很多项信息。有时候用户在输入信息的时候可能手误或者不小心输入了错误信息，对于有些重要的不可修改的项目(例如用于安全验证的邮箱、电话等信息)来说，小小的错误可能是致命的，所以我们习惯在一些提交重要信息的时候向用户打印一遍其输入的内容，确认以后再进行保存。

本实例就在用户注册的时候把用户提交的信息使用 ViewData 对象传递到 View 中，呈现出来，等待用户确认。

5.2.3 实例应用

【例 5-2】实现用户注册确认页面。

(1) 打开上一节使用的项目。

(2) 创建一个处理注册功能的 Controller，命名为 RegisterController。默认的 Index 动作，这里不需要修改。

(3) 为 RegisterController 创建对应的视图目录。在 Views 目录中创建一个名为 Register 的目录。

(4) 为 RegisterController 下的 Index 动作添加一个对应的视图文件 Index.aspx，保存在 Views/Register 目录下。该视图文件继承自第 5.1 节中创建的母版页。

Index.aspx 视图文件的主要代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
<div id="register">
<form action="/Register/Create" method="post">
   <table border="0" width="500">
   <thead><tr><td colspan="2">
      <h2>用户注册</h2>
   </td></tr></thead>
   <tbody>
   <tr><td class="td left">登录名：</td>
   <td><input name="loginName" /></td>
   </tr><tr>
   <td class="td_left">密码：</td><td><input name="passwrod" type="password"
/></td>
   </tr><tr>
   <td class="td left">确认密码：</td><td><input name="passwrod2"
type="password" /></td>
   </tr><tr>
   <td class="td_left">安全邮箱：</td>
   <td><input name="email" />输入你常用的邮箱(找回密码使用)</td>
   </tr><tr>
   <td class="td left">联系电话：</td>
   <td><input name="phone" />输入你常用的联系电话(或手机)</td>
   </tr><tr>
   <td colspan="2" align="center"><br /><button type="submit"> 提交
</button><br /><br /></td>
   </tr>
   </tbody>
   </table>
</form>
</div>
</asp:Content>
```

(5) 上面代码是一个注册表单，表单中有登录名、密码、邮箱和电话等项。为了封装这些数据，我们添加一个 Model 的 UserInfo 类，该类代码如下：

```
public class UserInfo
{
   public string LoginName { get; set; }
   public string Password { get; set; }
```

```
    public string Email { get; set; }
    public string Phone { get; set; }
}
```

(6) 上面表单中设置将数据提交到/Register/Create 上，所以需要在 RegisterConnection 中添加一个名为 Create 的 Action，代码如下：

```
public ActionResult Create()
{
    UserInfo user = new UserInfo()
    {
        LoginName = Request.Form["loginName"],
        Password = Request.Form["password"],
        Phone = Request.Form["phone"],
        Email = Request.Form["email"]
    };
    ViewData["UserInfo"] = user;

    return View();
}
```

上面代码将用户提交过来的数据封装到一个 UserInfo 对象中，然后将该对象保存到视图数据字典中。

(7) 为 Create 动作创建一个对应的视图页面 Create.aspx，用于呈现视图数据字典中的用户信息，页面的主要代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
<div id="register">
    <h2>用户注册</h2>
    <% MvcAppView1.Models.UserInfo user =
          ViewData["UserInfo"] as MvcAppView1.Models.UserInfo; %>

    你刚才提交的信息如下：<br />
    登 录 名：<%= user.LoginName %><br />
    登录密码：<%= "******" %><br />
    安全邮箱：<%= user.Email %><br />
    联系电话：<%= user.Phone %><br />
    <br />
    <button>确定注册</button>
    <a href="/Register">返回重填</a>
    <br /><br />
</div>
</asp:Content>
```

上面代码取出视图数据字典中的用户信息，进行类型转换，然后分别将各项信息呈现在页面中。这里密码当然不能直接明文显示，我们使用几个星号代替。

5.2.4　运行结果

运行程序，访问/Register/Index，结果如图 5-6 所示。

在注册页面中输入注册用户的信息，单击【提交】按钮，结果如图 5-7 所示。

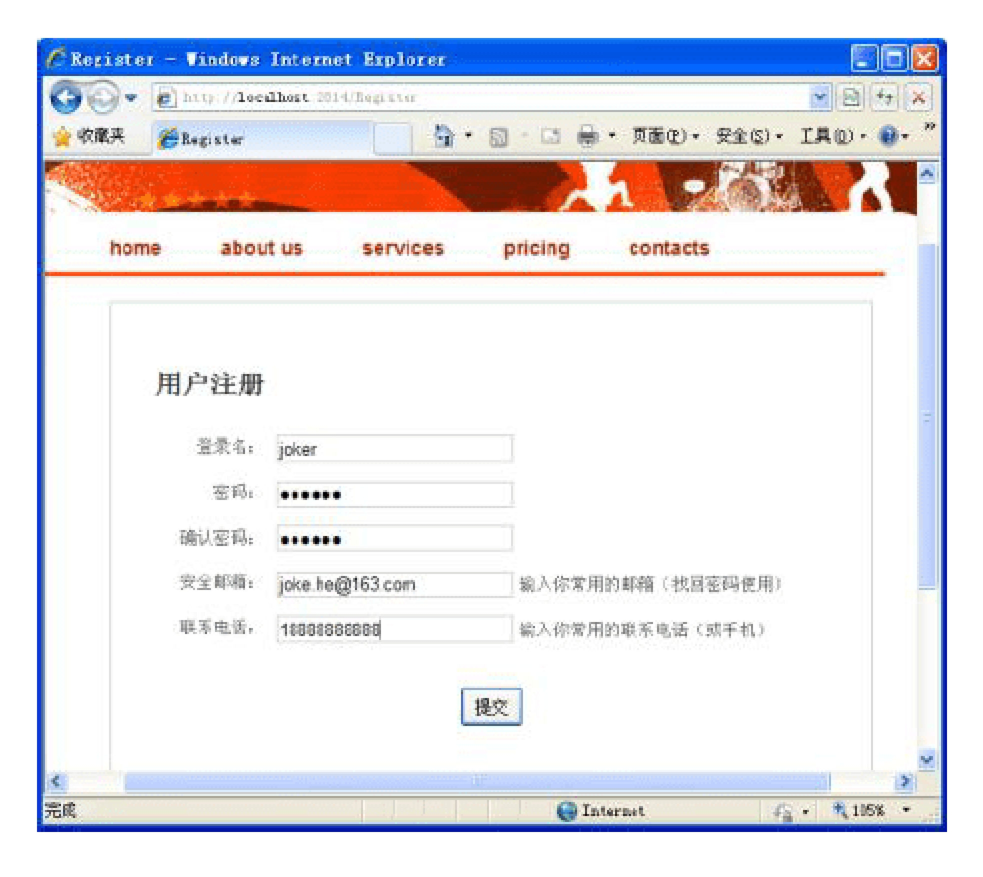


图 5-6　用户注册页面



图 5-7　注册信息确认页面

5.2.5　实例分析



源码解析

本实例接收用户提交的信息，并将其封装到一个 UserInfo 对象中，同时将 UserInfo 对象存储在视图数据字典中。

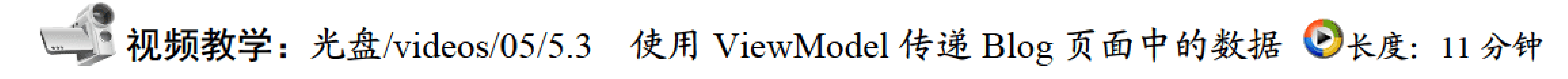
之后在视图页面取出视图数据字典中保存的 UserInfo 对象，并进行数据类型恢复。最后在页面中显示出对象中各个属性的值。

5.3　使用 ViewModel 传递 Blog 页面中的数据

上一节我们讲了如何使用视图数据字典将 Controller 处理好的数据呈现在页面中。或许大家已经发现了一些不太人性化的地方，那就是对所传递的数据的类型判断上。

视图数据字典 ViewData 和 TempData 对象都是弱类型的对象，使用的时候还需要进行数据类型的转换和验证。稍有不慎，就会酿成错误。

ASP.NET MVC 提供了一种强类型的数据传递方式——ViewModel，能很好地解决这个问题。



视频教学：光盘/videos/05/5.3　使用 ViewModel 传递 Blog 页面中的数据　长度：11 分钟

5.3.1　基础知识

在上一节中我们讲了 ASP.NET MVC 的两个数据字典对象 ViewData 和 TempData。如果要在页面中呈现一个对象，我们需要将这个对象添加到视图数据字典中，并在 View 中获取它。

例如上一节的实例，Controller 中的代码如下：

```
public ActionResult Create()
{
```

```
    UserInfo user = new UserInfo()
    {
        LoginName = Request.Form["loginName"],
        Password = Request.Form["password"],
        Phone = Request.Form["phone"],
        Email = Request.Form["email"]
    };
    ViewData["UserInfo"] = user;

    return View();
}
```

之后，我们可以在视中获取并显示它们：

```
<% MvcAppView1.Models.UserInfo user =
        ViewData["UserInfo"] as MvcAppView1.Models.UserInfo; %>
登 录 名：<%= user.LoginName %><br />
安全邮箱：<%= user.Email %><br />
联系电话：<%= user.Phone %><br />
```

因为视图数据字典返回的是一个弱类型的对象，所以在呈现该对象的时候有必要将其强制转换为相应类型的对象。如果 View 提供的视图具有与发送过来的模式相同的类型，那么代码将变得更加清晰。如此，强类型的数据传递方式 ViewModel 就应运而生了。

前面一节我们提到过 ViewPage 类被 ViewPage<TModel>类继承，而每个 aspx 视图又继承了 ViewPage<TModel>类，这一点在 View 文件中的@Page 指令中已明确说明。@Page 指令代码如下：

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
```



ascx 和 master 页面分别继承自 ViewUserControl<TModel>和 ViewMasterPage<TModel>类，这两个类也是强类型视图类。

要使用强类型的视图，可以在控制器方法中通过对 View()方法的重载来指定到 View 中的模型对象。

例如，我们要将一个用户信息传递到视图中，可以这样做：

```
public ActionResult Index()
{
    UserInfo user = new UserInfo()
    {
        LoginName = "admin",
        Password = "123",
        Email = "joke.he@163.com",
        Phone = "13888888888"
    };

    return View(user);
}
```

在这种情况下，我们将对象直接使用 View()方法传递到 View 中。

下一步，我们还需要修改视图的类型，也就是修改@Page 指令中标识该视图所继承的类的属性，例如：

```
<%@ Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<MvcAppView1.Models.UserInfo>" %>
```

如此，便可以在页面中直接使 Model 对象访问这个对象了，如图 5-8 所示。

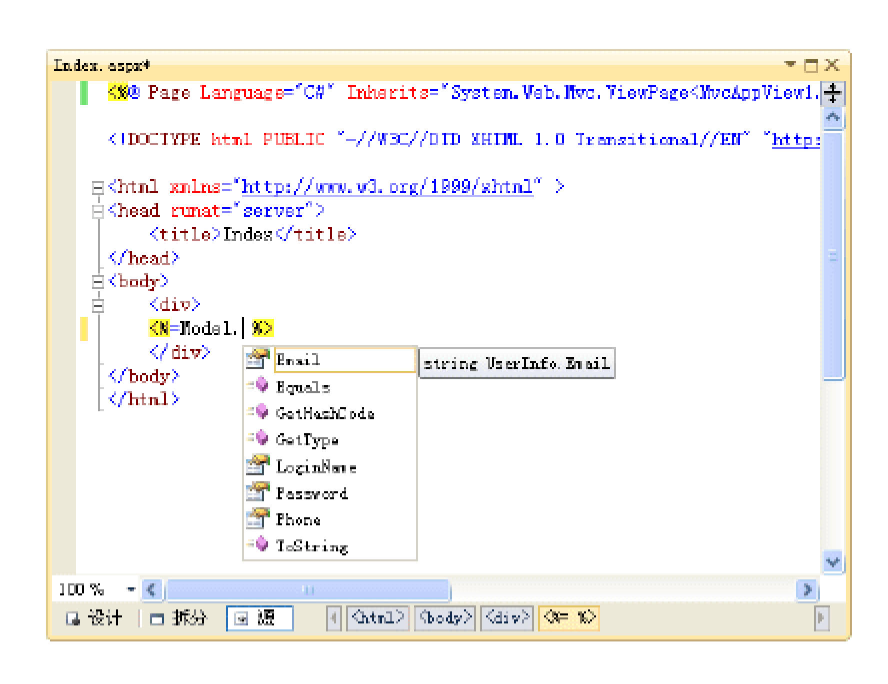


图 5-8　强类型视图的智能感应

这是 ASP.NET MVC 中使用强类型数据视图比较不错的一个方法。在视图中我们完全可以使用 Visual Studio 提供的智能感应功能来获取强类型的 Model。



通常页面中并不会只用一个数据对象，而 View()方法只能将一个对象发送到视图中。我们可以写一个对应视图的实体类，在类里将该视图用到的所有对象作为属性封装起来，就可以实现对多个数据对象的传递。

5.3.2　实例描述

通常，博客系统中会有标题、心情、文章分类列表以及文章等内容。这里有许多类型的数据，所以使用一个 ViewModel(视图模型)封装各部分数据，将会是一个不错的选择。

本实例我们就使用一个 BlogIndexView 类来封装页面中的各部分数据，并将其传递到页面中显示出来。

5.3.3　实例应用

【例 5-3】使用 ViewModel 传递 Blog 页面中的数据。

(1) 我们还是运用上一节中的项目。

(2) 既然要传递多部分数据，我们先创建一些封装数据的 ViewModel。在 Models 目录下创建一个 BlogModels.cs 文件，并修改该文件中的代码，创建 3 个实体类，主要代码如下：

```
namespace MvcAppView1.Models
{
    public class BlogIndexView
    {
        public string Title { get; set; }
        public string Mood { get; set; }
        public IList<BlogClass> BlogClasses{ get; set; }
        public BlogInfo Blog { get; set; }
    }
```

```
    public class BlogInfo
    {
        public string Title { get; set; }
        public string Details { get; set; }
    }

    public class BlogClass
    {
        public string Title { get; set; }
        public string Explain { get; set; }
    }
}
```

(3) 创建一个处理请求的 BlogController 类。在 Controller 目录中创建一个 Controller，命名为 BlogController。修改文件代码，主要代码如下：

```
public class BlogController : Controller
{
    public ActionResult Index()
    {
        BlogIndexView biv = new BlogIndexView()
        {
            Title = "zhh 的博客",
            Mood = "冬天就要来了……好像已经来了……"
        };
        biv.BlogClasses = new List<BlogClass>(){
            new BlogClass(){ Title="dotNET Framework", Explain="我对.NET 平台的
认知与感悟"},
            new BlogClass(){ Title="ASP.NET", Explain="ASP.NET 中的一些纠结" },
            new BlogClass(){ Title="ASP.NET MVC", Explain="ASP.NET 的新东西——
MVC" },
            new BlogClass(){ Title="个人笔记", Explain="学习中的一些感悟" }
        };
        biv.Blog = new BlogInfo() {
            Title = "第一个.NET 程序",
            Details = "<p>其他不说，来看代码：</p><blockquote><p><code>using
System;\n\n" +
            "namespace mdibb {\n\tclass Class1 {\n\t\tstatic void Main(string[]
args) {\n" +
            "\t\t\tConsole.WriteLine(\"Hello, World\");\n\t\t}\n\t}\n" +
            "}</code></p></blockquote><p>上面代码我向控制台中打印一个行文字“Hello
World”</p>" };

        return View(biv);
    }
}
```

(4) 创建一个对应 BlogController 中的 Index 动作的视图文件。在 Views 目录中创建一个名为 Blog 的文件夹，并在下面创建一个视图文件，名为 Index.aspx。

(5) 修改 Index.aspx 文件中的@Page 指令，如下所示：

```
<%@ Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<MvcAppView1.Models.BlogIndexView>" %>
```

(6) 修改 Index.aspx 文件的内容部分。

其中 head 部分代码如下：

```
<head runat="server">
    <title><%= Model.Title %></title>
    <link href="../../Content/Blog.css" rel="stylesheet" type="text/css" />
</head>
```

然后修改 body 部分，主要代码如下：

```
<div id="header">
    <h1><%=Model.Title %></h1>
    <p><%=Model.Mood %></p>
</div>
<div id="body">
    <div id="links">
        <p>
        <%foreach (var bc in Model.BlogClasses)
            { %>
            <a href="#"><%=bc.Title %></a><br/>
            <%=bc.Explain %>
            <br/><br/>
        <%} %>
        </p>
    </div>
    <div id="main">
        <h2><%=Model.Blog.Title %></h2>
        <%=Model.Blog.Details %>
    </div>
</div>
```

所有编码工作完成，下面我们来看效果。

5.3.4 运行结果

运行该应用程序。首先访问/Blog/Index，运行结果如图 5-9 所示。



图 5-9 博客页面运行结果

5.3.5　实例分析



源码解析

该实例将页面中需要使用的数据封装成一个 BlogIndexView 实体类，然后在 Action 中收集所有数据并封装到 BlogIndexView 类的实例中，再使用 View()方法的重载将该实例发送到视图中。在视图中使用 Model 属性访问 Action 发送过来的 BlogIndexView 类的实例，一一取出封装的数据，呈现到页面视图中。

5.4　常见问题解答

5.4.1　在 View 中能否操作 Model



在 View 中我能不能操作 Model 呢？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

在 ASP.NET MVC 中，都说要把视图和业务分离开，让视图中只展示数据。那么我在视图中能不能直接操作 Model 呢？

【解决办法】答案是肯定的。

不过 Model 也分两种，一种是实体 Model，存储的是一些实体类；另一种是业务 Model，存储的是一些业务逻辑处理程序(业务方法)。

在视图中我们当然可以访问实体类封装的对象，所以我们直接操作了 Model。这一点在前面的 MVC 结构示意图中也有所体现，图中也明确表示了 View(视图)依赖于 Model(模型)。

不过有一点需要注意：一般来说，在视图中是不允许操作业务模型的。因为业务模型处理的是业务逻辑，而我们通常将业务处理方法的调配权交由 Controller 控制，所以在 View 中不要调用 Model 的业务方法，否则就有违 MVC 三个模块划分的初衷了。

5.4.2　在 ASP.NET MVC 中能否使用 WebForm 服务器端控件



在 ASP.NET MVC 中还能使用 WebForm 服务器端控件吗？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

学了 MVC，看到视图又回归到那种“炸酱面”式的代码，有点怀念 Web 服务器端的控件，例如 Repeater、DataList 和 DetailsView 等。

我能不能在 View 中使用这些服务器端控件呢？

【解决办法】其实 ASP.NET MVC 和 ASP.NET WebForm 应用程序之间的区别就是视图的一些变化，ASP.NET WebForm 是将整个页面作为一个表单处理。实际上 ASP.NET MVC 使用的也是 ASP.NET WebForm 的页面机制，所以在 ASP.NET MVC 中同样可以使用 WebForm 的

服务器端控件。

但是，因为在 MVC 中我们不习惯给控件添加事件(其实也不可以)，所以尽量不要使用太复杂的页面控件，以免出现一些莫名其妙的错误。

这里建议处理重复信息的时候使用迭代控件 Repeater。

5.5 习 题

一、填空题

(1) 能被 Controller 自动识别的视图文件类型有 aspx 和__________。

(2) 在强类型视图中，可以使用__________对象访问视图动作传递过来的强类型数据。

(3) 如果我们要向视图传递一个强类型的对象，该使用什么方法？请补充下面代码。

```
public ActionResult Index()
{
    Student zhangsan = new Student()
    {
        //略
    };

                 ;
}
```

(4) 如果我们要跨页传递数据，可以使用__________对象。

二、选择题

(1) 将视图单独分离出来的好处有__________。

A. 可视化编辑，所见即所得

B. 程序开发简单

C. 不用维护

D. 可以自动生成

(2) 在 Views 目录中有一个所有 Controller 共享的目录，名字是__________。

A. Shared

B. Share

C. Common

D. GongXiang

(3) TempData 对象是一个__________类型的对象。

A. IDictionary

B. Dictionary

C. ITempDataDictionary

D. TempDataDictionary

(4) 下面关于 TempData 对象的说明，不正确的是__________。

A. TempData 对象可以跨页传递数据

B. TempData 对象是一个数据字典

C. TempData 对象需要类型转换

D. TempData 对象是一个强类型的数据传递对象

三、上机练习

上机练习：使用 ViewModel 实现个人信息修改页面。

通过本章的学习，大家应该了解 View 是如何从 Controller 中获取数据的。本练习我们就使用 ViewModel 强类型数据传递方式来从 Controller 将用户信息传递到视图中。页面执行结果如图 5-10 所示。



图 5-10　个人信息修改页面



第 6 章　页面辅助类

内容摘要

使用了 ASP.NET MVC，我们不能再使用页面控件，我们需要在页面中手动生成一些 HTML 代码。似乎又回到了古老的 ASP 时代，就连写一个下拉列表框都要写一个循环和一堆乱七八糟的表达式来实现，这对于用惯了 ASP.NET WebForm 控件的我们来说非常痛苦。

不过，真的没有其他方法了吗？当然不是。

ASP.NET 还提供了一些在视图中起辅助作用的静态类和方法，可以很方便地使用服务器端代码来生成一些页面元素。

本章主要讲如何使用 ASP.NET MVC 的默认视图引擎提供的两个辅助类 HtmlHelper 和 URLHelper。

学习目标

- 了解为什么要用 HtmlHelper 类
- 掌握 HtmlHelper 类的表单生成方法
- 掌握 HtmlHelper 类的超链接生成方法
- 掌握 HtmlHelper 类的局部视图加载方法
- 掌握 URL 辅助类 URLHelper

6.1 页面辅助类 HtmlHelper

相对于 ASP.NET WebForm，ASP.NET MVC 架构的应用程序有一个显著的特点：它为我们提供了对应用程序完全控制的能力，其中包括 HTML 标记。

不过，完全控制真的很好吗？有人宣称这是 ASP.NET MVC 架构的第一大优点，确实是这样吗？

完全控制确实不错，也是相对于 ASP.NET WebForm 架构应用程序的一个显著特征，但是这种情况的好坏还要取决于应用程序的具体情况。

很多时候我们并不需要对 HTML 标记进行控制，因为我们不需要关心其具体的形状。因为应用程序错误处理的不同，往往具体的运行中我们才能发现 HTML 出现的异常。所以虽然完全控制不错，但也在无形中增加了许多职责。让你不得不怀念 ASP.NET WebForm 中的服务器控件。

HTML 辅助方法提供了介于完全控制和 ASP.NET WebForm 中间的控件。这些方法包含在 ASP.NET MVC 架构中，它能够帮助我们呈现一些常用的标记。



视频教学：光盘/videos/06/6.1 页面辅助类 长度：7 分钟

6.1.1 HtmlHelper 类

在 ASP.NET MVC 中，ViewPage 类包含了一个名为 Html 的属性，该属性是一个 HtmlHelper 类的对象，该对象用于在视图中呈现 HTML 元素。

HtmlHelper 类位于 System.Web.Mvc.Html 命名空间，该类本身的功能并不多，其主要功能由一系列静态扩展类组成。它们分别是 FormExtensions 类、InputExtensions 类、LinkExtensions 类、SelectExtensions 类、TextAreaExtensions 类、ValidationExtensions 类和 RenderPartialExtensions 类等。在 Visual Studio 对象浏览器中，我们可以查看到各个扩展类的实现方法详情，如图 6-1 所示。



在对象浏览器中我们可以看到系统为 HtmlHelper 类添加了十多个扩展类，限于篇幅我们只讲其中常用的 7 个。

在 ASP.NET MVC 视图文件中，我们可以使用 ViewPage 类的 Html 属性直接访问上面各类的扩展方法，如图 6-2 所示。

HtmlHelper 类通过扩展方法实现了 ASP.NET MVC 中常用的各种控件。同样，如果我们不喜欢这些辅助方法，可以添加自己的 Html 辅助类来实现这些功能，也可以编写 HtmlHelper 类的扩展方法来实现一些特定需要的功能。

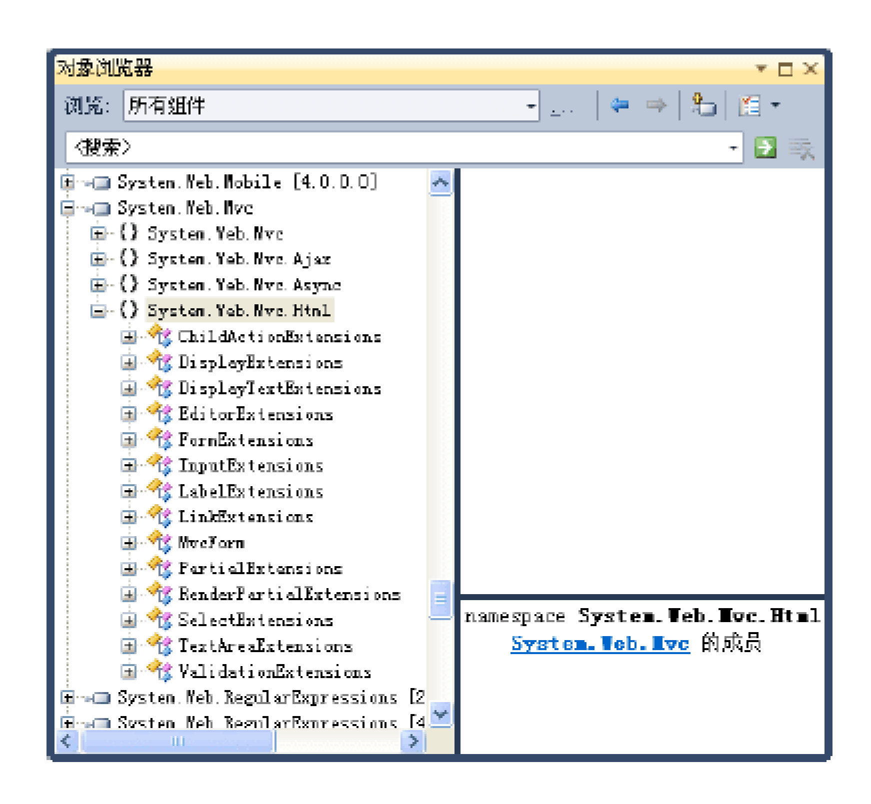


图 6-1　对象浏览器

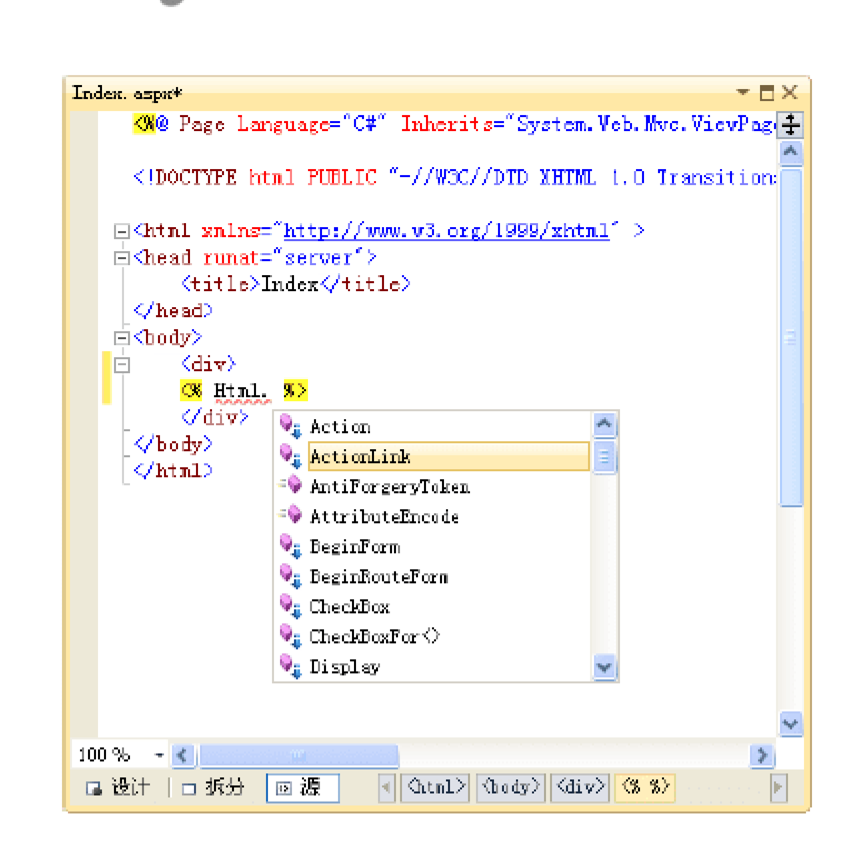


图 6-2　在视图中使用 Html 辅助对象

6.1.2　为什么使用 Html 辅助方法

对于这个问题，其实最简洁的回答就是“它能帮我们干活”。确实，它能够帮我们做很多我们不想做的工作。

例如在 Action 中已经写好了一个集合对象，代码如下：

```
public ActionResult Index()
{
    IList<int> level = new List<int> { 1, 2, 3, 4, 5 };
    ViewData["Level"] = level;

    return View();
}
```

我们要使用这个集合在视图中呈现一个下拉列表框，传统方法需要在页面中使用 foreach 或 for 循环遍历这个集合，取出数据，如下所示：

```
<select name="level">
<% foreach (var level in ViewData["Level"] as IList<int>)
    { %>
    <option value="level"><%= level %></option>
<% } %>
</select>
```

如果使用 Html 辅助方法，就省去了页面中大段的遍历代码，只需要一句话就可以打印出来一个下拉列表框。

这里我们需要先简单地修改一个 Action 中的代码，如下所示：

```
public ActionResult Index()
{
    IList<int> level = new List<int> { 1, 2, 3, 4, 5 };
    ViewData["Level"] = new SelectList(level);

    return View();
}
```

我们对这个集合使用 SelectList 简单地包装一下，就可以在视图中直接使用了。代码如下：

```
<%= Html.DropDownList("Level") %>
```

这一行就完成了向页面中输入一个下拉列表框的功能，其好处在此不多说了。

不过，在 ASP.NET MVC 的官方教程中有一些关于 Helper 的特性在这里要强调一下。

- 所有的辅助方法都会给元素属性的值编码。
- 所有的辅助方法都会给页面中显示的值编码，例如链接文本。
- 接收 RouteValueDictionary 的辅助方法含有一个对应的重载，它允许指定一个匿名对象为字典。
- 同样，接收 IDictionary<string, object>的辅助方法也有一个相对应的重载，它允许将匿名对象指定为字典。
- 用于呈现为表单字段的辅助方法在 ModelState 字典中自动搜索它们当前的值。辅助方法的名称和参数将作为搜索字典的键。
- 如果 ModelState 包含一个错误，与之相关的表单辅助方法将呈现一个 input-validation-error(输入验证错误)的 CSS 类和任何指定的 CSS 类。在项目模板包含的默认样式表 style.css 中，有该类的样式声明。

6.2 使用动态表单上传个性头像

在需要交互的页面中，最核心的对象就是 form(表单)。一般应用程序必须使用它向服务器提交数据与上传文件。

一般的 HTML 表单通常由<form>标记开始，到</form>标记结束，中间可能穿插任意多个 HTML 元素。所有需要向服务器提交的页面表单元素，都必须放在这两个标记中间。

在 ASP.NET MVC 中，对 HtmlHelper 类提供 form 支持的扩展类就是 FormExtension 类。

 **视频教学：**光盘/videos/06/6.2 使用动态表单上传个性头像(1) 长度：5 分钟
使用动态表单上传个性头像实例(2) 长度：8 分钟

6.2.1 基础知识

FormExtensions 类是一个静态类，其声明在命名空间 System.Web.Mvc.Html 中，其中定义了 3 个扩展方法。开发人员可以使用它们在视图中设置表单和表单路由的定义，这 3 个方法分别是 BeginForm、BeginRouteForm 和 EndForm。

1. BeginForm()方法

BeginForm()方法实现表单的开始部分，也就是说它将组织一个<form>标记及其属性。在当前版本的 MVC 中，该方法有 13 个重载。

重载方法：

- BeginForm()
- BeginForm(object routeValues)
- BeginForm(RouteValueDictionary routeValues)

- BeginForm(string actionName, string controllerName)
- BeginForm(string actionName, string controllerName, object routeValues)
- BeginForm(string actionName, string controllerName, RouteValueDictionary routeValues)
- BeginForm(string actionName, string controllerName, FormMethod method)
- BeginForm(string actionName, string controllerName, object routeValues, FormMethod method)
- BeginForm(string actionName, string controllerName, RouteValueDictionary routeValues, FormMethod method)
- BeginForm(string actionName, string controllerName, FormMethod method, object htmlAttributes)
- BeginForm(string actionName, string controllerName, FormMethod method, Idictionary <string, object> htmlAttributes)
- BeginForm(string actionName, string controllerName, object routeValues, FormMethod method, object htmlAttributes)
- BeginForm(string actionName, string controllerName, RouteValueDictionary routeValues, FormMethod method, IDictionary<string, object> htmlAttributes)

BeginForm()方法的各个重载中主要包含以下参数。

1) object routeValues

路由参数的值，一个包含路由参数的对象。通过检查对象的属性，利用反射检索参数。此对象通常是使用对象初始值设定项语法创建的。

例如我们向当前页面提交一个参数 id 的值 23，可以使用以下代码完成：

```
<%= Html.BeginForm(new { id = 23 }) %>
```

上面代码将生成下面的 HTML 标记。

```
<form action="/Test/Index/23" method="post">
```

2) RouteValueDictionary routeValues

路由参数的值，一个包含路由参数的集合，表示“键/值”对形式。

例如视图中有以下代码：

```
<%
   IDictionary<string,object> args=new Dictionary<string,object>();
   args.Add("id",23);
%>
<%= Html.BeginForm(new RouteValueDictionary(args)) %>
```

同样会生成下面的 HTML 标记：

```
<form action="/Test/Index/23" method="post">
```

3) string actionName

form 中要提交到 URL 的 Action 名称。

4) string controllerName

form 中要提交到 URL 的 Controller 名称。

例如我们要生成一个 action 属性为/Home/Index 的 form 表单，可以这样编写代码：

```
<%= Html.BeginForm("Index", "Home"); %>
```

上面代码将生成这样的 HTML 标记：

```
<form action="/Home" method="post">
```



因为 Index 动作被 URLRouting 配置为默认动作，所以这里生成的 action 属性只有一个 Controller 名称/Home。

5) FormMethod method

指明在用户提交表单时，将使用哪个方法来处理该表单的数据。可选值为 FormMethod.Post 或 FormMethod.Get。

例如以下代码：

```
<%= Html.BeginForm("Index", "Home",FormMethod.Get) %>
```

将生成下面的 HTML 元素：

```
<form action="/Home" method="get">
```

6) object htmlAttributes

一个包含要为该 HTML 元素设置 HTML 特性的对象，该属性可以是一个匿名对象，而且可以在该匿名对象中设置相应的属性值。

例如我们要设置 form 表单的 enctype 属性值为 multipart/form-data，可以使用以下代码：

```
<%= Html.BeginForm("Index", "Home",FormMethod.Post, new { enctype =
"multipart/form-data" }) %>
```

上面代码执行后的结果如下：

```
<form action="/Home" enctype="multipart/form-data" method="post">
```

7) IDictionary<string, object> htmlAttributes

一个包含要为该 HTML 元素设置 HTML 特性的集合，可以设置该元素相应的属性值。

例如下面的代码：

```
<%
   IDictionary<string, object> args = new Dictionary<string, object>();
   args.Add("enctype", "multipart/form-data");
%>
<%= Html.BeginForm("Index", "Home", FormMethod.Post,args )%>
```

同样可以生成下面代码：

```
<form action="/Home" enctype="multipart/form-data" method="post">
```

2．BeginRouteForm()方法

BeginRouteForm()方法将根据 URL 路由规则实现一个指定路径的<form>标记。当前版本的 MVC 中该方法有 12 个重载。

重载方法：

- BeginRouteForm(object routeValues)

- BeginRouteForm(RouteValueDictionary routeValues)
- BeginRouteForm(string routeName)
- BeginRouteForm(string routeName, object routeValues)
- BeginRouteForm(string routeName, RouteValueDictionary routeValues)
- BeginRouteForm(string routeName, FormMethod method)
- BeginRouteForm(string routeName, object routeValues, FormMethod method)
- BeginRouteForm(string routeName, RouteValueDictionary routeValues, FormMethod method)
- BeginRouteForm(string routeName, FormMethod method, object htmlAttributes)
- BeginRouteForm(string routeName, FormMethod method, IDictionary<string, object> htmlAttributes)
- BeginRouteForm(string routeName, object routeValues, FormMethod method, object htmlAttributes)
- BeginRouteForm(string routeName, RouteValueDictionary routeValues, FormMethod method, IDictionary<string, object> htmlAttributes)

BeginRouteForm()方法的各个重载主要包括以下几个参数。

- object routeValues　路由参数的值，可以是一个匿名对象。
- RouteValueDictionary routeValues　路由参数的值的集合。
- string routeName　生成 URL 的路由名称，即 URLRouting 里的 Name 参数，例如系统默认的 Default。
- FormMethod method　提交请求的方法。
- object htmlAttributes　生成 HTML 元素的属性，可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　生成 HTML 元素的属性，这里是一个属性集合。



因为各属性的参数前面已经有所了解，所以这里就不再重复讲解了。本章以后的其他扩展方法也是如此。

3．EndForm()方法

EndForm()方法比效简单，它只负责生成表单定义的结束部分。

例如我们在视图中设置如下代码：

```
<%= Html.EndForm() %>
```

这行代码将生成如下 HTML 语句：

```
</form>
```

因为在表单中，结束标记</form>不需要进行属性配置，所以这里的 EndForm()方法也没有任何带参数的重载形式。

在实际应用程序开发中，我们还可以使用 using 语句来操作 BeginForm()方法，而不需要书写这里的 EndForm()方法。

例如视图中有以下代码：

```
<% using (Html.BeginForm())
   {%>
<%} %>
```

运行以后将生成以下 HTML 语句：

```
<form action="/" method="post"></form>
```

怎么样，是不是更加简洁了？

6.2.2 实例描述

前面章节中已经演示过如何向服务器提交注册信息。很多时候，在一些交互性强的 Web 系统中，通常可以让用户上传一些个性头像来突显个人特色。

本实例就来演示如何使用 Form 表单向服务器上传一个个性头像。

6.2.3 实例应用

【例 6-1】使用动态表单上传个性头像。

(1) 新建一个空的 MVC 项目。

(2) 首先在 Controller 目录中添加一个控制器类，命名为 UploadController。修改该 Controller 中的代码，如下所示：

```
public class UploadController : Controller
{
    public string FacePic = "/Upload/default face.gif";

    public ActionResult Index()
    {
        ViewData["FacePic"] = FacePic;
        return View("Index");
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult UploadFace()
    {
        foreach (string fn in Request.Files)
        {
            HttpPostedFileBase hpf = Request.Files[fn];
            if (!this.HasFile(hpf)) continue;     //如果没有包含文件，继续下一次循环
            this.FacePic = this.SaveFile(hpf);    //保存文件，获取文件名
            break;
        }
        return Index();
    }

    private string SaveFile(HttpPostedFileBase hpf)
    {
        string path = Server.MapPath("~/Upload/");
        hpf.SaveAs(path + hpf.FileName);
        return "/Upload/" + hpf.FileName;
```

```
    }

    private bool HasFile(HttpPostedFileBase hpf)
    {
        return hpf != null && hpf.ContentLength > 0;
    }
}
```

上面代码中的 Index 方法用于在第一次请求的时候显示上传头像界面，UploadFace 方法用于接收上传请求，保存上传的图片并显示到页面中。另外两个方法是上传功能的辅助方法。

(3) 这里我们需要一个上传页面视图文件，命名为 Index.aspx。修改页面视图代码，主要代码如下：

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
<div id="register">
 <% using (Html.BeginForm("UploadFace", "Upload", FormMethod.Post, new
{ enctype = "multipart/form-data" }))
    { %>
    <table border="0" width="500">
    <thead><tr><td colspan="2">
        <h2>上传头像</h2>
    </td></tr></thead>
    <tbody>
    <tr><td class="td_left"> </td>
    <td>
        <img alt="默认头像" src="<%= ViewData["FacePic"] %>"
            style="width: 160px; height: 160px; border:solid 1px #ccc; " /></td>
    </tr><tr>
    <td class="td left">选择头像：</td><td><input type="file" name="face"
style="width:350px;" /></td>
    </tr><tr>
    <td colspan="2" align="center"><br /><button type="submit"> 上传
</button><br /><br /></td>
    </tr>
    </tbody>
    </table>
    <%= ViewData["Flag"] %>
<%} %>
</div>
</asp:Content>
```

这里使用 using 语句包装了 Html.BeginForm()方法，将自动生成一组 form 标记。在 form 表单中，显示一个头像图片，接收一个视图字典的值，另外还有一个文件框，用于上传图片。



这里需要注意，要上传文件，我们必须把表单的 enctype 属性设置为 multipart/form-data，表示上传的是二进制数据，否则文件上传将会失败。

下面运行一下代码看看效果。

6.2.4　运行结果

运行该应用程序。访问/Upload/Index 路径，结果如图 6-3 所示。

单击【选择头像】文本框右侧的【浏览】按钮，选择一个图片文件，再单击【上传】按钮，执行上传操作，上传成功以后，页面效果如图 6-4 所示。



图 6-3　上传页面



图 6-4　上传成功页面

6.2.5　实例分析



源码解析

本实例使用 using 语句封装了 Html.BeginForm()方法，生成一个 form 表单。该表单将请求提交到 Upload 控制器的 UploadFace 动作上，请求方式为 POST，同时为表单元素添加一个 enctype 属性，属性值为 multipart/form-data，表示上传的是二进制数据。在服务器端接收到数据以后进行保存，并将保存的文件路径和文件名传递到页面中，再将上传的图片显示出来。

6.3　使用页面辅助类扩展用户注册功能

表单中通常需要提交一些文本或其他类型的数据，所以大多数的表单离不开文本框控件。文本框控件在 HTML 中表示为 input 标记，该标记可以通过 type 属性将其设置成文本框、密码框、单选按钮、多选按钮或隐藏域等。

在 ASP.NET MVC 中提供了一个 input 标记扩展功能的类 InputExtensions，下面我们就来详细了解一下。

视频教学：光盘/videos/06/6.3　使用页面辅助类扩展用户注册功能(1)　长度：12 分钟
使用页面辅助类扩展用户注册功能实例(2)　长度：10 分钟

6.3.1　基础知识

InputExtensions 类是一个静态类，其声明在命名空间 System.Web.Mvc.Html 中。其中定义的扩展方法分别对多选按钮(CheckBox)、隐藏域(Hidden)、密码框(Password)、单选按钮(RadioButton)和文本框(TextBox)等进行了封装。开发人员可以使用它们在视图中创建相应的表

单元素。

1. CheckBox()方法

CheckBox()方法封装了 HTML 元素中的多选框控件。其定义了 6 个重载方法供开发人员使用。重载方法：

- CheckBox(string name)
- CheckBox(string name, bool isChecked)
- CheckBox(string name, bool isChecked, object htmlAttributes)
- CheckBox(string name, object htmlAttributes)
- CheckBox(string name, IDictionary<string, object> htmlAttributes)
- CheckBox(string name, bool isChecked, IDictionary<string, object> htmlAttributes)

这些重载方法的参数如下。

- string name　表单字段的名称。
- bool isChecked　该复选框初始状态是否已被选中。
- object htmlAttributes　该复选框 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　该复选框 HTML 元素的属性值，该值是一个数据字典。

例如在视图中有以下代码：

```
<%= Html.CheckBox("MyCheckBox1",true,new{ id="checkBox1" }) %>
```

将生成如下 HTML 语句：

```
<input checked="checked" id="checkBox1" name="MyCheckBox1" type="checkbox"
value="true" /><input name="MyCheckBox1" type="hidden" value="false" />
```

2. Hidden()方法

Hidden()方法封装了 HTML 元素中的隐藏域控件。其定义了 4 个重载方法供开发人员使用。重载方法：

- Hidden(string name)
- Hidden(string name, object value)
- Hidden(string name, object value, object htmlAttributes)
- Hidden(string name, object value, IDictionary<string, object> htmlAttributes)

这些重载方法的参数如下。

- string name　窗体字段的名称或用于查找值的 System.Web.Mvc.ViewDataDictionary 键
- object value　隐藏的 input 元素的值。如果此值为 null，则从 System.Web.Mvc. ViewDataDictionary 对象检索该元素的值。如果该对象中不存在任何值，则从 System.Web.Mvc.ModelStateDictionary 对象检索该值。
- object htmlAttributes　隐藏域 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　隐藏域 HTML 元素的属性值，该值是一个数据字典。

例如在视图中有以下代码：

```
<%= Html.Hidden("Hidden1","This is Hidden Value") %>
```

则将在 HTML 代码中生成以下语句：

```
<input id="Hidden1" name="Hidden1" type="hidden" value="This is Hidden Value"
/>
```

3. Password()方法

Password()方法封装了 HTML 元素中的密码输入框控件，该方法有 4 个重载形式。

重载方法：

- Password(string name)
- Password(string name, object value)
- Password(string name, object value, object htmlAttributes)
- Password(string name, object value, IDictionary<string, object> htmlAttributes)

该方法的参数与 Hidden 方法一致，这里不再详细说明。

例如我们的视图文件中有以下代码：

```
<%= Html.Password("pwd1", 123456) %>
```

上面代码将在页面中生成如下的 HTML 代码：

```
<input id="pwd1" name="pwd1" type="password" value="123456" />
```

4．RadioButton()方法

RadioButton()方法封装了 HTML 元素中的单选按钮控件，该方法有 6 个重载形式。

重载方法：

- RadioButton(string name, object value)
- RadioButton(string name, object value, object htmlAttribute)
- RadioButton(string name, object value, IDictionary<string, object> htmlAttribute)
- RadioButton(string name, object value, bool isChecked)
- RadioButton(string name, object value, bool isChecked, object htmlAttribute)
- RadioButton(string name, object value, bool isChecked, IDictionary<string, object> htmlAttribute)

这些重载方法的参数如下。

- string name　窗体字段的名称和用于查找值的 System.Web.Mvc.ViewDataDictionary 键。
- object value　如果选择此单选按钮，则为在发送窗体时提交的此单选按钮的值。如果 System.Web.Mvc.ViewDataDictionary 或 System.Web.Mvc.ModelStateDictionary 对象中选定的单选按钮的值与此值匹配，则选择此单选按钮。
- object htmlAttribute　一个对象，其中包含要为该元素设置的 HTML 特性。
- IDictionary<string, object> htmlAttribute　单选按钮 HTML 元素的属性值，该值是一个数据字典。
- bool isChecked　单选按钮的选择状态。如果要选择单选按钮，则为 true；否则为 false。

例如我们的视图文件中有以下代码：

```
<%= Html.RadioButton("radioButton1", "buttonValue", true)%>
```

上面代码将在页面中生成如下的 HTML 语句：

```
<input checked="checked" id="radioButton1" name="radioButton1" type="radio"
value="buttonValue" />
```

5．TextBox()方法

TextBox()方法封装了 HTML 元素中的文本框控件，该方法有 4 个重载形式。

重载方法：

- TextBox(string name)
- TextBox(string name, object value)
- TextBox(string name, object value, object htmlAttribute)
- TextBox(string name, object value, IDictionary<string, object> htmlAttribute)

该方法的参数与 Hidden()和 Password()方法一致，这里不再详细说明。

例如我们的视图文件中有以下代码：

```
<%= Html.TextBox("TextBox1", "TextValue", new { maxlength = 3 })%>
```

上面代码将在页面中生成如下的 HTML 语句：

```
<input id="TextBox1" maxlength="3" name="TextBox1" type="text"
value="TextValue" />
```

6.3.2　实例描述

在第 6.2 节中，我们使用了普通的纯 HTML 标签的形式编写了一个注册表单。这种方式不太容易进行控制。

本实例我们将使用 ASP.NET MVC 提供的页面辅助类对该注册表单进行重构，并对注册信息添加一些扩展属性。

6.3.3　实例应用

【例 6-2】使用页面辅助类扩展用户注册功能。

(1) 运用上一节的项目。

(2) 首先打开 RegisterController，在该控制器中添加一个 Action，命名为 Reg，代码如下：

```
public ActionResult Reg()
{
    return View();
}
```

(3) 添加一个对应 Reg 动作的视图，命名为 Reg.aspx。添加页面代码，如下所示：

```
<% using (Html.BeginForm("Create", "Register", FormMethod.Post))
   { %>
    <table border="0" width="500">
    <thead><tr><td colspan="2">
        <h3>用户注册</h3>
    </td></tr></thead><tbody>
    <tr><td class="td_left">登录名：</td>
```

```
        <td><%=Html.TextBox("loginName") %></td>
    </tr><tr><td class="td left">密码：</td>
        <td><%=Html.Password("password") %></td>
    </tr><tr><td class="td left">确认密码：</td>
        <td><%=Html.Password("password2") %></td>
    </tr><tr><td class="td left">性别：</td>
        <td><%=Html.RadioButton("sex", true, new { style = "border:0;
width:30px;" })%>男
            <%=Html.RadioButton("sex", false, new { style = "border:0;
width:30px;" })%>女</td>
    </tr><tr><td class="td left">已婚：</td>
        <td><%=Html.CheckBox("married", false, new { style = "border:0;
width:30px;" })%></td>
    </tr><tr><td class="td left">安全邮箱：</td>
        <td><%=Html.TextBox("email") %>输入你常用的邮箱(找回密码使用)</td>
    </tr><tr><td class="td left">联系电话：</td>
        <td><%=Html.TextBox("phone") %>输入你常用的联系电话(或手机)</td>
    </tr><tr>
    <td colspan="2" align="center"><br /><button type="submit"> 提交
</button><br /><br /></td>
    </tr>
    </tbody>
    </table>
<%} %>
```

在上面的代码中，我们使用页面辅助类生成了一个页面表单。因为本节演示的是页面辅助类，所以这里将其他业务逻辑代码省略。

6.3.4 运行结果

运行该项目，访问/Register/Reg 路径，结果如图 6-5 所示。

在页面空白处右击，然后从弹出的快捷菜单中选择【查看源文件】命令，结果如图 6-6 所示。

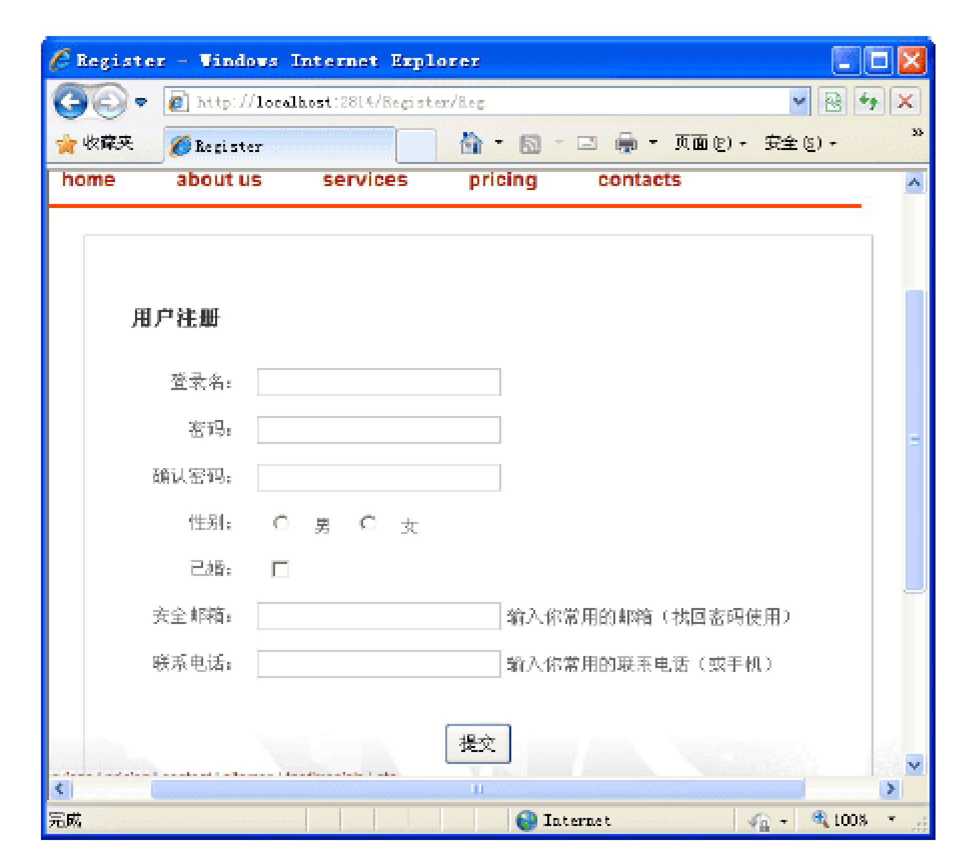


图 6-5　用户注册页面



图 6-6　注册页面源代码

6.3.5　实例分析



源码解析

本实例使用页面辅助类 HtmlHelper 下的扩展方法生成的表单元素来对注册页面进行重构。使用 Html.TextBox()方法替换了页面中 HTML 标记<input>，然后使用 Html.RadioButton()和 Html.CheckBox()方法封装了新添加的两个用户注册属性。

另外，本实例还使用了上一节讲过的 Html.BeginForm()方法来对页面中的 Form 表单进行了封装。

6.4　超链接扩展类

Web 应用程序少不了超链接。ASP.NET MVC 专门使用 LinkExtensions 类向 Htmlhelper 类扩展了两个静态方法 ActionLink()和 RouteLink()，可以让开发者在视图中自由地设置相关的链接。

下面分别对它们予以说明。



视频教学：光盘/videos/06/6.4　超链接扩展类　　　　长度：12 分钟

6.4.1　ActionLink()方法

ActionLink()方法封装了一个超链接，其中定义了 10 个重载的扩展方法供开发人员使用。

重载方法：

- ActionLink(string linkText, string actionName)
- ActionLink(string linkText, string actionName, object routeValues)
- ActionLink(string linkText, string actionName, object routeValues, object htmlAttributes)
- ActionLink(string linkText, string actionName, RouteValueDictionary routeValues)
- ActionLink(string linkText, string actionName, RouteValueDictionary routeValues, IDictionary<string, object> htmlAttributes)
- ActionLink(string linkText, string actionName, string controllerName)
- ActionLink(string linkText, string actionName, string controllerName, object routeValues, object htmlAttributes)
- ActionLink(string linkText, string actionName, string controllerName, RouteValueDictionary routeValues, IDictionary<string, object> htmlAttributes)
- ActionLink(string linkText, string actionName, string controllerName, string protocol, string hostname,string fragment, object routeValues, object htmlAttributes)

- ActionLink(string linkText, string actionName, string controllerName, string protocol, string hostname,string fragment, RouteValueDictionary routeValues, IDictionary<string, object> htmlAttributes)

这些重载方法的参数如下。

- string linkText　指定元素的内部文本。
- string actionName　动作的名称
- object routeValues　一个包含路由参数的匿名对象。通过检查对象的属性，可利用反射检索参数。
- RouteValueDictionary routeValues　一个包含路由参数的集合对象。
- object htmlAttributes　该 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　该 HTML 元素的属性值，该值是一个数据字典。
- string controllerName　控制器的名称。
- string protocol　URL 协议，例如 http 或 https。
- string hostname　URL 的主机名称。
- string fragment　URL 的片段名称，指定一个锚点名称。

例如我们可以使用下面语句来生成一个超链接：

```
<%= Html.ActionLink("Home New","New","Home")%>
```

该语句生成的 HTML 代码如下：

```
<a href="/Home/New">Home New</a>
```

假如我们要生成一个指定完整 URL 的超链接，可以使用如下代码：

```
<%= Html.ActionLink("Home New", "New", "Home","http","www.12345.com","a", new
{ id = 32 }, new { style = "text-decoration:none;" })%>
```

上面代码生成的结果如下：

```
<a href="http://www.12345.com:2814/Home/New/32#a"
style="text-decoration:none;">Home New</a>
```



在这里生成的结果中，是以当前 Web 应用程序所使用的端口为目标 URL 端口。

6.4.2　RouteLink()方法

RouteLink()方法也将封装一个视图中的超链接，与 ActionLink()方法不同的是，它可以根据路由配置规则来生成匹配指定路由的超链接。

RouteLink()方法定义了 11 个重载的扩展方法供开发人员使用。

重载方法：

- RouteLink(string linkText, object routeValues)

- RouteLink(string linkText, RouteValueDictionary routeValues)
- RouteLink(string linkText, string routeName)
- RouteLink(string linkText, string routeName, object routeValues)
- RouteLink(string linkText, string routeName, RouteValueDictionary routeValues)
- RouteLink(string linkText, object routeValues, object htmlAttributes)
- RouteLink(string linkText, RouteValueDictionary routeValues, IDictionary<string,object> htmlAttributes)
- RouteLink(string linkText, string routeName, object routeValues, object htmlAttributes)
- RouteLink(string linkText, string routeName, RouteValueDictionary routeValues, IDictionary <string,object> htmlAttributes)
- RouteLink(string linkText, string routeName, string protocol, string hostName, string fragment, object routeValues, object htmlAttributes)
- RouteLink(string linkText, string routeName, string protocol, string hostName, string fragment, RouteValueDictionary routeValues, IDictionary<string,object> htmlAttributes)

这些重载方法的参数如下。

- string linkText　指定元素的内部文本。
- string routeName　路由的名称。
- object routeValues　一个包含路由参数的匿名对象。通过检查对象的属性，可利用反射检索参数。
- RouteValueDictionary routeValues　一个包含路由参数的集合对象。
- object htmlAttributes　该 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string,object> htmlAttributes　该 HTML 元素的属性值，该值是一个数据字典。
- string protocol　URL 协议，例如 http 或 https。
- string hostname　URL 的主机名称。
- string fragment　URL 的片段名称，指定一个锚点名称。

例如我们可以使用下面语句来生成一个超链接：

```
<%= Html.RouteLink("NewLink", new { controller="Home", action="New" }) %>
```

该语句生成的 HTML 代码如下：

```
<a href="/Home/New">NewLink</a>
```

也可以使用该方法来实现更复杂的链接配置，如下：

```
<%= Html.RouteLink("NewLink", "Default", new { controller = "Home", action =
"New", id = 32 }, new { style = "text-decoration:none;" })%>
```

上面的代码将生成一个 HTML 元素，如下所示：

```
<a href="/Home/New/32" style="text-decoration:none;">NewLink</a>
```

6.5 使用局部视图处理站点搜索模块

在 ASP.NET WebForm 应用程序中有个用户控件机制，可以让我们把一些多个页面公用的模块独立起来存放到一个文件中，以便实现程序模块化和代码复用的功能。

在 ASP.NET MVC 中，沿用了 ascx 文件，但它仅作为页面视图的角色而存在。那么如何能像 WebForm 中那样使用用户控件呢？

答案就是 PartialView(局部视图)。

**视频教学：**光盘/videos/06/6.5 使用局部视图处理站点搜索模块 长度：7 分钟

6.5.1 基础知识

一般来说，PartialView(局部视图)在 ASP.NET MVC 中也是以用户控件(扩展名 asca)的形式存在的。



虽然在 ASP.NET MVC 的视图中可以使用 Web 页面(aspx)充当局部视图，但是因为其包含有完整的页面结构，所以一般不建议使用 Web 页面作为局部视图。

在 ASP.NET MVC 中，可以使用 RenderPartialExtensions 类来扩展 HtmlHelper 方法。

RenderPartialExtensions 类中包含一个 RenderPartial()方法，该方法用来引入一个局部视图。RenderPartial()方法有 4 个重载形式。

重载方法：

- RenderPartial(string partialViewName)
- RenderPartial(string partialViewName, ViewDataDictionary viewData)
- RenderPartial(string partialViewName, object model)
- RenderPartial(string partialViewName, object model, ViewDataDictionary viewData)

这些重载方法的参数如下。

- string partialViewName 局部视图的名称，这里不包含扩展名。
- object model 用于局部视图的模型。如果该局部视图是一个强类型视图，则需要为其指定该项。
- ViewDataDictionary viewData 局部视图使用的数据字典。



默认情况下，局部视图和页面视图使用的是同一个 ViewData，所以可以不用为其指定具体的视图数据字典。

6.5.2 实例描述

搜索模块通常是一个网站中最常用的功能模块，它一般需要放在站点的多个页面中。按照以前 ASP.NET WebForm 的习惯，通常将它作为一个用户控件存放。

本实例我们就使用 ASP.NET MVC 的局部视图来处理搜索模块。

6.5.3　实例应用

【例 6-3】使用局部视图处理站点搜索模块。

(1) 打开前面我们使用的项目。

(2) 先来添加一个 Controller，命名为 PartialController。同时系统会自动为其添加一个名为 Index 的 Action，这里我们不需要修改它们。

(3) 在 Views 目录中创建一个名为 Partial 的文件夹。

(4) 在 Partial 目录下添加一个局部视图文件，名为 Search.ascx。编辑该文件，添加搜索功能的代码，如下所示：

```
<%@ Control Language="C#"
Inherits="System.Web.Mvc.ViewUserControl<dynamic>" %>

<h2>搜索</h2>
<input type="text" name="s" id="searchtext" />
<input type="submit" id="searchsubmit" value="搜索" />
```

(5) 在 Partial 目录下添加一个视图文件，名为 Index.aspx。编辑该页面，添加页面内容。在使用搜索功能的地方用 Html 对象的 RenderPartial()方法引入前面创建的局部视图，如下所示：

```
…
<div id="secondary">
   <% Html.RenderPartial("Search"); %>
    <h2>关于</h2>
…
```

6.5.4　运行结果

运行该项目，访问/Partial/Index 路径，结果如图 6-7 所示。

在该页面中，右击页面空白处，选择【查看源文件】命令。打开该页面的源文件，可以看到，系统自动将局部视图中的代码引入进来，如图 6-8 所示。



图 6-7　引入局部视图的运行结果

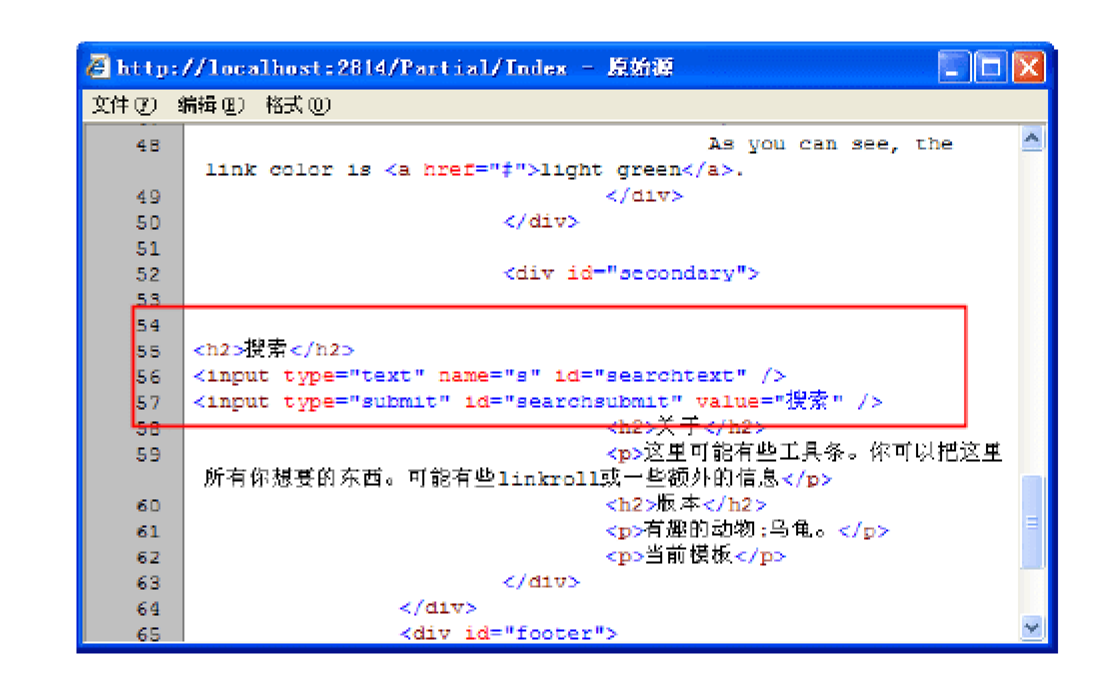


图 6-8　页面源文件

6.5.5 实例分析



源码解析

本实例将页面中的搜索功能模块单独存放，以实现应用程序代码的复用。在页面中需要用到的地方使用 HtmlHelper 的扩展方法 RenderPartial()将其引入即可。

该方法的使用效果与 ASP.NET WebForm 中的用户控件的效果一样。

6.6 完善注册页面

列表框控件是 HTML 表单中使用较为普遍也是十分复杂的控件。通常我们需要编写一个循环，将其展示到页面中。

在 ASP.NET WebForm 中提供了一个 DropDownList 控件，使用非常方便。同样，ASP.NET MVC 的 HtmlHelper 也扩展了一些实现类似功能的方法，就是这里的 DropDownList()和 ListBox()方法。

**视频教学：**光盘/videos/06/6.6 完善注册列表　　长度：9 分钟

6.6.1 基础知识

在 ASP.NET MVC 中，SelectExtensions 类为 HtmlHelper 类提供了一些静态方法，用于封装一些列表选择控件，例如这里的 DropDownList()和 ListBox()方法。

1. DropDownList()方法

DropDownList()方法主要实现一个下拉列表框。其中定义了 8 个重载方法供开发人员使用。重载方法：

- DropDownList(string name)
- DropDownList(string name, string optionLabel)
- DropDownList(string name, IEnumerable<SelectListItem> selectList)
- DropDownList(string name, IEnumerable<SelectListItem> selectList, object htmlAttributes)
- DropDownList(string name, IEnumerable<SelectListItem> selectList, IDictionary<string,object> htmlAttributes)
- DropDownList(string name, IEnumerable<SelectListItem> selectList, string optionLabel)
- DropDownList(string name, IEnumerable<SelectListItem> selectList, string optionLabel, object htmlAttributes)
- DropDownList(string name, IEnumerable<SelectListItem> selectList, string optionLabel, IDictionary<string,object> htmlAttributes)

这些重载方法的参数如下。

- string name　要返回的表单中元素的名称。
- string optionLabel　默认为第一个空项的文本，此参数可以为 null。
- IEnumerable<SelectListItem>selectList　一个用于填充下拉列表的 System.Web.Mvc. SelectListItem 对象的集合。
- object htmlAttributes　该 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　该 HTML 元素的属性值，该值是一个数据字典。

例如在视图中有如下代码：

```
<%= Html.DropDownList("List", new SelectList(new int[] { 2, 4, 6, 3, 2 }))%>
```

将在页面中生成一个下拉列表框，HTML 代码如下：

```
<select id="List" name="List"><option>2</option>
<option>4</option>
<option>6</option>
<option>3</option>
<option>2</option>
</select>
```

2．ListBox()方法

ListBox()方法将在页面中生成一个列表框。其中定义了 4 个重载方法供开发人员使用。

重载方法：

- ListBox(string name)
- ListBox(string name, IEnumerable<SelectListItem> selectList)
- ListBox(string name, IEnumerable<SelectListItem> selectList, object htmlAttributes)
- ListBox(string name, IEnumerable<SelectListItem> selectList, IDictionary<string,object> htmlAttributes)

这些重载方法的参数如下。

- string name　要返回的表单中元素的名称。
- IEnumerable <SelectListItem>selectList　一个用于填充列表的 System.Web.Mvc.Select-ListItem 对象的集合。
- object htmlAttributes　该 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　该 HTML 元素的属性值，该值是一个数据字典。

例如在视图中有如下代码：

```
<%= Html.ListBox("List", new SelectList(new string[] { "郑州", "石家庄", "济南
", "武汉" }))%>
```

将在页面中生成一个下拉列表框，HTML 代码如下：

```
<select id="List" multiple="multiple" name="List"><option>郑州</option>
<option>石家庄</option>
<option>济南</option>
<option>武汉</option>
```

```
</select>
```

6.6.2 实例描述

从前面的讲述我们知道，使用 DropDownList()和 ListBox()方法可以很方便地实现一个选择框控件。在实际应用中，我们经常用到这些列表选择控件，例如在注册信息的时候使用的国家和地区、日期、血型以及星座等属性都可以使用列表框控件来实现。

本实例我们就使用一个列表框控件来完善我们的用户注册页面，这里我们为其添加一个星座选择项。

6.6.3 实例应用

【例 6-4】完善注册页面。

(1) 运用前面我们使用的项目。

(2) 先修改 RegisterController 中的 Reg 方法，准备一个下拉列表框的数据。代码如下：

```
public ActionResult Reg()
{
   IDictionary<int, string> star = new Dictionary<int, string>();
   star.Add(1, "白羊座");
   star.Add(2, "金牛座");
   star.Add(3, "双子座");
   star.Add(4, "巨蟹座");
   star.Add(5, "狮子座");
   star.Add(6, "处女座");
   star.Add(7, "天秤座");
   star.Add(8, "天蝎座");
   star.Add(9, "射手座");
   star.Add(10, "摩羯座");
   star.Add(11, "水瓶座");
   star.Add(12, "双鱼座");
   SelectList starList = new SelectList(star, "Key", "Value");
   ViewData["star"] = starList;
   return View();
}
```

(3) 修改页面中的代码。在注册表单中间添加一个【星座】下拉列表框。代码如下：

```
…
</tr><tr><td class="td left">确认密码：</td>
   <td><%=Html.Password("password2") %></td>
</tr><tr><td class="td left">星座：</td>
   <td><%=Html.DropDownList("star") %></td>
</tr><tr><td class="td left">性别：</td>
   <td><%=Html.RadioButton("sex", true, new { style = "border:0;
width:30px;" })%>男
…
```

好了，可以运行项目看看结果了。

6.6.4　运行结果

运行我们的项目，访问/Register/Reg 路径，结果如图 6-9 所示。

右击页面中的空白处，选择【查看源文件】命令，打开一个【原始源】窗口，如图 6-10 所示。

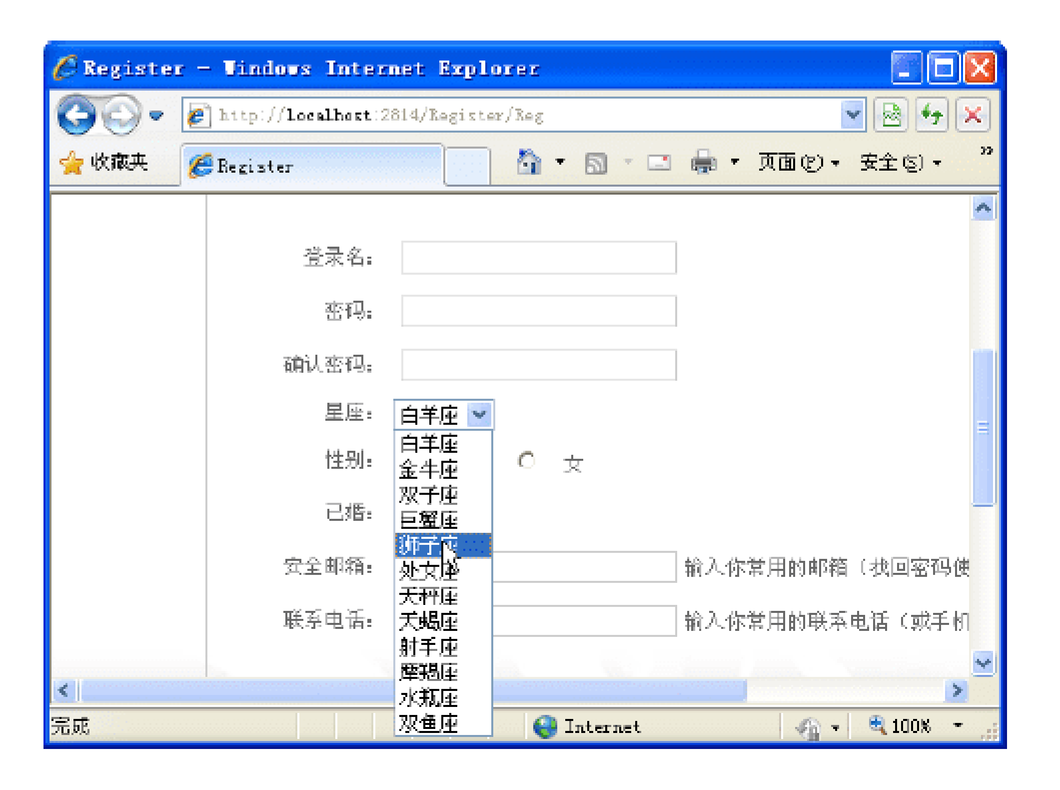


图 6-9　注册页面的【星座】下拉列表

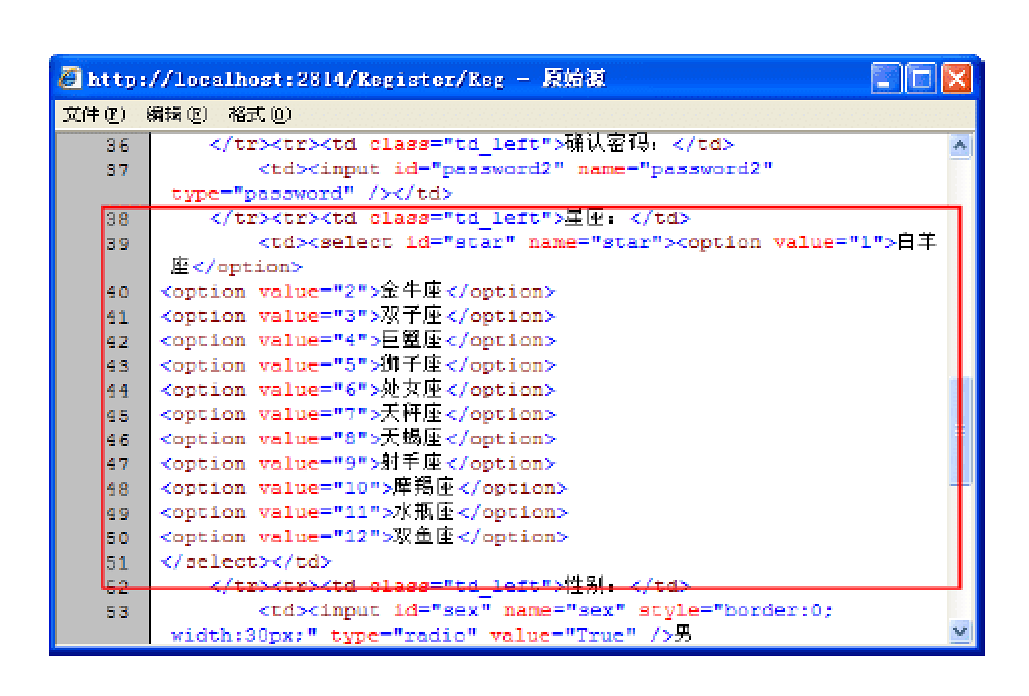


图 6-10　生成下拉列表框的 HTML 代码

6.6.5　实例分析



源码解析

本实例首先在 Action 中声明了一个字典(IDictionary<int, string>)，用于存储星座信息，然后使用 SelectList 类的构造方法将其封装成一个 SelectList 类型的列表对象，并把该对象存放到视图字典中，最后使用 HtmlHelper 类的 DropDownList()方法在页面中展示一个下拉列表框控件。由于这里没有设置列表项和默认值，所以该方法将根据 name 参数的值到视图字典中搜索列表项并自动绑定。

6.7　文本域扩展类



视频教学：光盘/videos/06/ 6.7　文本域扩展类　　　长度：4 分钟

在 HTML 表单中，文本域控件(<textarea></textarea>)也是一个常用的表单组件，通常用于输入大段的文本信息。

ASP.NET MVC 中的 HtmlHelper 也对其进行了封装，使我们可以在服务器端更方便地控制文本域的结构。

ASP.NET MVC 中的 TextAreaExtensions 类对 HtmlHelper 类进行了扩展，使其对文本域控件提供了支持。

TextAreaExtensions 类中定义了一个 TextArea()方法，对页面的 textarea 元素进行了封装。该方法有 8 个重载形式。

重载方法：

- TextArea(string name)
- TextArea(string name, object htmlAttributes)
- TextArea(string name, IDictionary<string,object> htmlAttributes)
- TextArea(string name, string value)
- TextArea(string name, string value, object htmlAttributes)
- TextArea(string name, string value, IDictionary<string,object> htmlAttributes)
- TextArea(string name, string value, int rows, int columns, object htmlAttributes)
- TextArea(string name, string value, int rows, int columns, IDictionary<string,object> htmlAttributes)

这些重载方法的参数如下。

- string name　要返回的表单中元素的名称。
- object htmlAttributes　该 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　该 HTML 元素的属性值，该值是一个数据字典。
- string value　页面中表单元素的值(内容)。
- int rows　生成的文本域的行数。
- int columns　生成的文本域的列数。

例如在视图中有如下代码：

```
<%= Html.TextArea("myTextArea", "<h1>这是一个标题</h1><p>这是一段文本</p>", 3, 50,
new { style = "border: solid 1px #ccc;" })%>
```

将在页面中生成一个文本域，HTML 代码如下：

```
<textarea cols="50" id="myTextArea" name="myTextArea" rows="3" style="border:
solid 1px #ccc;">
&lt;h1&gt;这是一个标题&lt;/h1&gt;&lt;p&gt;这是一段文本&lt;/p&gt;</textarea>
```

6.8 登 录 验 证

在 ASP.NET WebForm 中，验证机制为我们带来很大方便。HtmlHelper 也提供了一些扩展方法，可以实现在页面中显示 ModelState 中存在的错误信息，以便开发人员在处理应用程序执行时的错误。

**视频教学：**光盘/videos/06/6.8　登录验证　　　　长度：9 分钟

6.8.1 基础知识

ValidationExtensions 类里提供了两个静态方法 ValidationMessage()和 ValidationSummary()，

这两个方法实现了将 ModelState 字典中的信息显示到页面中的功能。

1. ValidationMessage()方法

ValidationMessage()方法可以将 ModelState 字典中指定的错误信息显示到视图中，并且可以使用 CSS 控制其风格。该方法有 6 个重载形式。

重载方法：

- ValidationMessage(string modelName)
- ValidationMessage(string modelName, object htmlAttributes)
- ValidationMessage(string modelName, string validationMessage)
- ValidationMessage(string modelName, string validationMessage, object htmlAttributes)
- ValidationMessage(string modelName, IDictionary<string,object> htmlAttributes)
- ValidationMessage(string modelName, string validationMessage, IDictionary<string,object >htmlAttributes)

这些重载方法的参数如下。

- string modelName　所验证的属性或模型对象的名称。
- string validationMessage　要在指定字段包含错误时显示的消息。
- object htmlAttributes　该 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　该 HTML 元素的属性值，该值是一个数据字典。

例如在控制器动作中我们将一个错误添加到 ModelState 中：

```
public ActionResult Index()
{
    var ms = new ModelState();
    ms.Errors.Add("ErrorMessage.");
    ModelState["Err"] = ms;

    return View();
}
```

接下来在视图中使用 ValidationMessage()方法显示该错误信息，如下所示：

```
<%= Html.ValidationMessage("Err") %>
```

这样将在页面中显示如下 HTML 代码：

```
<span class="field-validation-error">ErrorMessage.</span>
```

当然，如果在 ModelState 中没有找到指定键的值，什么也不输出。

另外，我们还可以在视图中重写该错误信息，例如：

```
<%= Html.ValidationMessage("Err", "This is a Error!") %>
```

将在页面中生成如下 HTML 代码：

```
<span class="field-validation-error">This is a Error!</span>
```

2. ValidationSummary()方法

ValidationSummary()方法将在页面中显示一个 ModelState 字典中所有确认错误的未经排序

的列表，而且可以使用 CSS 控制其风格。该方法有 8 个重载形式。

重载方法：

- ValidationSummary()
- ValidationSummary(bool excludePropertyErrors)
- ValidationSummary(string message)
- ValidationSummary(bool excludePropertyErrors, string message)
- ValidationSummary(string message, object htmlAttributes)
- ValidationSummary(bool excludePropertyErrors, string message, object htmlAttributes)
- ValidationSummary(string message, IDictionary<string,object> htmlAttributes)
- ValidationSummary(bool excludePropertyErrors, string message, IDictionary<string, object> htmlAttributes)

这些重载方法的参数如下。

- string message　要在 ModelState 中包含错误时显示的消息标题。
- bool excludePropertyErrors　true 表示使摘要仅显示模型级别的错误，false 表示使摘要显示所有错误。
- object htmlAttributes　该 HTML 元素的属性值，该值可以是一个匿名对象。
- IDictionary<string, object> htmlAttributes　该 HTML 元素的属性值，该值是一个数据字典。

例如在控制器动作中我们将一些错误添加到 ModelState 中：

```
public ActionResult Index()
{
    var modelState = new ModelState();
    modelState.Errors.Add("LoginName Error.");
    ModelState["LoginName"] = modelState;

    modelState = new ModelState();
    modelState.Errors.Add("Password Error.");
    ModelState["Password"] = modelState;

    return View();
}
```

之后在页面中使用本节讲的两个方法获取这些信息：

```
<%= Html.ValidationMessage("LoginName") %>
<%= Html.ValidationMessage("Password") %>
<%= Html.ValidationSummary("This is Title") %>
```

上面代码最终将在页面中生成如下 HTML 代码：

```
    <span class="field-validation-error">LoginName Error.</span>
    <span class="field-validation-error">Password Error.</span>
    <div class="validation-summary-errors"><span>This is Title</span>
<ul><li>LoginName Error.</li>
<li>Password Error.</li>
</ul></div>
```

6.8.2　实例描述

提起“验证”，使用得最多的就是权限验证吧。一般所有的信息管理系统都需要一个用户登录功能，以便系统能够很好地控制登录用户的权限。

既然需要验证，就说明有不合法的信息。我们在处理错误信息的时候往往使用浏览器弹出一个对话框，这样很不友好，但向页面中指定位置显示信息也不太方便。

在 ASP.NET MVC 中，我们可以使用 HtmlHelper 提供的验证信息机制在 Action 里控制错误信息的显示。

本实例我们就使用 HtmlHelper 的方法来实现用户登录中的错误提示。

6.8.3　实例应用

【例 6-5】登录验证。

(1) 运用前面我们使用的项目。

(2) 在 Controller 目录下添加一个 Controller，命名为 LoginController。

(3) 修改 LoginController，即修改其中代码如下：

```
public class LoginController : Controller
{
    public ActionResult Index()
    {
        if (Request.HttpMethod == "POST")
        {
            string username = Request.Form["Username"];
            string password = Request.Form["Password"];

            if (ValidateUser(username, password))
            {
                return View("Success");
            }
        }

        return View();
    }

    private bool ValidateUser(string username, string password)
    {
        bool flag = true;
        if (username == null || username == "")
        {
            ModelState modelState = new ModelState();
            modelState.Errors.Add("用户名输入错误！");
            ModelState["Username"] = modelState;
            flag = false;
        }
```

```
        if (password == null || password == "")
        {
            ModelState modelState = new ModelState();
            modelState.Errors.Add("密码输入错误！");
            ModelState["Password"] = modelState;
            flag = false;
        }
        return flag;
    }
}
```

(4) 在 Views 目录下创建一个名为 Login 的文件夹。
(5) 在 Login 文件夹中添加一个视图，命名为 Index，视图文件名称为 Index.aspx。
(6) 修改 Index 视图，添加代码，主要代码如下：

```
<div style="width:100%; ">
    <% using (Html.BeginForm())
       { %>
    <table border="0" cellpadding="0" cellspacing="0" style=" width:500px;
margin:auto;">
    <thead>
    <tr><td colspan="3"><h3>管理登录</h3></td></tr>
    </thead><tbody>
    <tr>
        <td class="td_left">登录名：</td>
        <td class="td_control"><%= Html.TextBox("Username")%></td>
        <td><%=Html.ValidationMessage("Username") %> </td>
    </tr><tr>
        <td class="td_left">密码：</td>
        <td class="td_control"><%=Html.Password("Password")%></td>
        <td><%=Html.ValidationMessage("Password") %> </td>
    </tr>
    <tr><td colspan="2" align="center"><br /><button type="submit">登录
</button></td></tr>
    </tbody>
    </table>
    <%} %>
</div>
```

6.8.4 运行结果

运行程序，访问/Login/Index 路径。

在打开的登录页面中直接单击【登录】按钮，在输入框后面的单元格中将显示错误提示信息，运行结果如图 6-11 所示。

图 6-11　登录验证错误信息

6.8.5　实例分析



源码解析

本实例将在页面表单提交的时候获取用户输入的信息，进行判断(这里只是演示，只判断是否为空)，如果某一项不合法，将在 ModelState 中添加一个错误信息。

最后在页面的适当位置使用 HtmlHelper 类的 ValidationMessage()方法来在视图中呈现 ModelState 中保存的错误信息。

6.9　URL 辅助类 URLHelper

前面讲过页面辅助类 HtmlHelper，它可以在页面中动态地生成一些 HTML 元素。ASP.NET MVC 还提供了一个专门生成 URL 的辅助类 URLHelper。

因为有些开发人员不习惯使用组件来生成超链接，所以经常会使用纯 HTML 的方式添加一个超链接标记，然后只要生成一个 URL 即可。有些地方需要直接显示一个 URL，URLHelper 就填补了这方面的空白。

UrlHelper 提供了常用的 4 个方法：Action()、Content()、Encode()和 RouteUrl()，用于根据不同的参数使用不同的方式生成 URL。



视频教学：光盘/videos/06/6.9　URL 辅助类　　　　长度：6 分钟

6.9.1　Action()方法

Action()方法根据传入的值生成 URL，该方法有 8 个重载形式供开发人员使用。

重载方法：

- Action(String actionName)

- Action(String actionName, Object routeValues)
- Action(String actionName, RouteValueDictionary routeValues)
- Action(String actionName, String controllerName)
- Action(String actionName, String controllerName, Object routeValues)
- Action(String actionName, String controllerName, RouteValueDictionary routeValues)
- Action(String actionName, String controllerName, Object routeValues, String protocol)
- Action(String actionName, String controllerName, RouteValueDictionary routeValues, String protocol, String hostName)

这些重载方法的参数如下。

- string actionName　控制器动作的名称。
- string controllerName　控制器的名称。
- Object routeValues　一个包含路由参数的对象。通过检查对象的属性，可以利用反射检索参数，取得参数的值。
- RouteValueDictionary routeValues　一个包含路由参数的集合。
- string protocol　URL 中的协议名称，例如 http 或 https。
- string hostname　URL 中的的主机名。

例如在视图中有如下代码：

```
<a href="<%= Url.Action("Create", "Product") %>">Create</a>
```

将会在页面中生成如下 HTML 语句：

```
<a href="/Product/Create">Create</a>
```

另外，我们也可以使用这种方式来生成一个 form(表单)，代码如下：

```
<form action="<%= Url.Action("Create", "Product", new { id = 1 } ) %>"
method="post">
</form>
```

生成的结果如下：

```
<form action="/Product/Create/1" method="post">
</form>
```

6.9.2　Content()方法

利用 Content()方法可以将一个相对路径转换为一个绝对路径。该方法只有一个参数，并且没有其他的重载形式。其格式如下：

```
Content(String contentPath)
```

Content()方法接收一个相对路径为参数，生成一个站点的绝对路径。

例如在视图中有如下代码：

```
<a href='<%= Url.Content("~/Product/Create")%>'>Create</a>
```

将会在页面中生成如下一行 HTML 语句：

```
<a href='/Product/Create'>Create</a>
```

6.9.3　Encode()方法

Encode()方法将把 URL 中的特殊字符编码成可以在 URL 中传递的密文形式。

Encode()方法只有一个参数，并且没有其他重载形式。其格式如下：

```
Encode(String url)
```

Encode()方法接收一个未加密的 URL 内容作为参数，该方法将对这个参数进行加密。

例如在视图中有如下代码：

```
<a href="<%= Url.Action("Create", "Product", new { id = Url.Encode("这是一段中文&内容") }) %>">Create</a>
```

将在页面中生成如下 HTML 代码：

```
<a
href="/Product/Create/%25e8%25bf%2599%25e6%2598%25af%25e4%25b8%2580%25e6%25ae%25b5%25e4%25b8%25ad%25e6%2596%2587%2526%25e5%2586%2585%25e5%25ae%25b9">Create</a>
```

6.9.4　RouteUrl()方法

RouteUrl()方法将根据指定的路由及参数生成一个 URL。该方法提供了 7 个重载形式供开发人员使用。

重载方法：

- RouteUrl(Object routeValues)
- RouteUrl(RouteValueDictionary routeValues)
- RouteUrl(String routeName)
- RouteUrl(String routeName, Object routeValues)
- RouteUrl(String routeName, RouteValueDictionary routeValues)
- RouteUrl(String routeName, Object routeValues, String protocol)
- RouteUrl(String routeName, RouteValueDictionary routeValues, String protocol, String hostName)

这些重载方法的参数如下。

- Object routeValues　一个包含路由参数的对象。通过检查对象的属性，可以利用反射检索参数，取得参数的值。
- RouteValueDictionary routeValues　一个包含路由参数的集合。
- String routeName　用于生成 URL 的路由的名称。
- string protocol　URL 中的协议名称，例如 http 或 https。
- string hostname　URL 中的的主机名。

例如在视图中有如下代码：

```
<a href="<%= Url.RouteUrl(new { controller = "Product", action = "Show", id =
"12" })%>">Show</a>
```

将在页面中生成如下的 HTML 文本：

```
<a href="/Product/Show/12">Show</a>
```

6.10 常见问题解答

6.10.1 Html.RenderPartial 报错



Html.RenderPartial 报错
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

执行当前 Web 请求期间，出现未处理的异常。请检查堆栈跟踪信息，以了解有关该错误以及代码中导致错误的详细信息。

异常详细信息 System.NullReferenceException：未将对象引用设置到对象的实例。

源错误:

行 24: </div>--%>

行 25: <% Html.RenderPartial("Header.ascx"); %>

行 26: </div>

源文件：e:\gz_project\ Visual Studio 2008\Design\Design.Web\U\UserDomains\Home.aspx 行:26

堆栈跟踪:

[NullReferenceException：未将对象引用设置到对象的实例。]

……

(以下省略)

【解决办法】如果你的 partial(本例中是 Header.ascx)是在当前请求的 controller 下(位于目录/Views/nameController 下)或共享目录下(/Views/Shared)，那么只要把后缀.ascx 去掉就行了，也就是把第 25 行换成：

```
<% Html.RenderPartial("Header.ascx"); %>
```

如果你的 partial 位于其他位置，那么你需要引用虚拟目录，目录依你的项目而定，形式如下：

```
<%: Html.Partial("~/Views/Shared/Partials/MyOtherPartial.ascx") %>
```

这个时候需要.ascx 后缀。如果不明白，请补充问题，希望我能帮到你。

6.10.2 为什么 ASP.NET MVC 要使用 BeginForm



为什么 ASP.NET MVC 要使用 BeginForm?
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

为什么 ASP.NET MVC 要使用 BeginForm？为什么不直接用<form></form>？

【解决办法】action 的 URL 可能受 UrlRouting 的影响而改变。

例如你将{controller}/{action}规则改为{action}/{controller}.html，那么你是不是要将所有页面的<form action="data/get"改为<form action="get/data.html"呢？

如果使用 Helper 就解决了这个问题。

其实所有的 Helper 无非就是解决程序中有变化的东西，例如绑定数据、URLRouting 的 URL 或相对路径，如果你的地址一直不会变化，完全可以使用 HTML 标签。

6.11　习　　题

一、填空题

(1) HtmlHelper 类位于__________命名空间。

(2) FormExtensions 类为 HtmlHelper 类扩展了 3 个方法，分别是__________、BeginRouteForm 和 EndForm。

(3) URLHelper 类提供了一个__________方法来将文本进行 URL 编码。

(4) 使用 HtmlHelper 类的__________方法，可以实现一个下拉列表框。

二、选择题

(1) 在 HtmlHelper 类中，使用__________方法可以生成一个文本域对标记。

A. TextArea()　　B. Action()

C. ListBox()　　D. TextBox()

(2) 使用一个无参的 Html.BeginForm()方法，可以生成一个__________的表单头标签。

A. 提交到网站首页　　B. 提交到当前页面

C. 提交到上级目录　　D. 没有提交目标

(3) 下面代码中__________可以生成如下一段 HTML 标记：

```
<select id="List" name="List"><option>1</option>
  <option>2</option>
  <option>3</option>
  <option>4</option>
</select>
```

A. `<%=Html.DropDownList("List",new int[]{ 1, 2, 3, 4 }) %>`

B. `<%=Html.ListBox("List",new int[]{ 1, 2, 3, 4 }) %>`

C. `<%=Html.ListBox("List", new SelectList(new int[] { 1, 2, 3, 4 }))%>`

D. `<%=Html.DropDownList("List",new SelectList(new int[]{ 1, 2, 3, 4 })) %>`

(4) 在 ASP.NET MVC 中，__________类提供了生成文本框、复选框、单选按钮和隐藏域等控件的方法。

A. InputExtensions

B. TextAreaExtensions

C. LinkExtensions

D. SelectExtensions

三、上机练习

上机练习： 使用 HtmlHelper 实现一个详细信息登记页面。

本章我们花了大量篇幅讲解了 HtmlHelper 的各个方法的功能，上机使用 HtmlHelper 生成一个实现详细信息登记功能的 Form 表单，如图 6-12 所示。



图 6-12　详细信息登录页面



第7章　在 View 中使用 WebForm 控件

内容摘要

随着 MVC 模式在 Java 中如火如荼地发展，Microsoft 也渐渐感觉到 MVC 的优势，也加入了 MVC 模式的应用行列。

Microsoft 的传统 Web 开发模式 ASP.NET WebForm 确实开创了 Web 程序的一个新纪元。它很容易上手，使开发人员很轻松地做一些原本很复杂的实现(例如 GridView 控件、DataList 控件和 Repeater 控件)。

对于这几年才开始发展的 ASP.NET MVC，因为在 UI 层中需要将 Controller 和 View 完全分离，所以它必须颠覆传统的 WebForm 开发模式中的事件驱动模式，使用传统的表单提交的 UI 处理思想。在 ASP.NET MVC 中，基本上摒弃了 ASP.NET WebForm 中的服务器端控件以及与之相关的事件处理机制。

但是因为 ASP.NET MVC 正在发展，所以其对开发中的实际应用功能的支持并不完善，例如集合的迭代，我们还需要使用传统的 foreach 遍历方式来实现，很不美观。不过，经过测试，在 ASP.NET MVC 的视图中，仍然可以使用 ASP.NET WebForm 中的服务器端控件来实现集合数据的迭代。

本章我们就来介绍服务器端迭代控件在 ASP.NET MVC 中的应用。

学习目标

- 为什么要在视图中使用服务器端控件
- 使用 Repeater 控件展示数据集合
- 使用 DataList 控件展示数据集合

7.1　迭代显示一个员工信息列表

俗话说：知己知彼，百点不殆。

要在 ASP.NET MVC 中使用迭代控件，必须先弄明白为什么需要使用它们，了解一下当前开发中遇到的问题，然后才能体会到我们为什么要破例违规使用 ASP.NET WebForm 控件。

下面就看看如何使用 foreach 的方式迭代一个集合。

**视频教学：**光盘/videos/07/7.1　迭代显示一个员工信息列表　　长度：7 分钟

7.1.1　实例描述

随着公司规模的扩大，员工管理就成了一个难题，经常见人力资源部的人来回各个部门统计问题，非常费力，所以公司决定开发一套人力资源管理系统。

现在公司里有一个 ASP.NET 小组时间比较充足，所以使用 ASP.NET MVC 技术来开发这个项目，也能顺便锻炼一下员工的能力，两全其美。

下面这个实例就拿人力资源管理系统中的员工列表功能给大家展示一下 ASP.NET MVC 中的数据集合迭代的实现。

7.1.2　实例应用

【例 7-1】迭代显示一个员工信息列表。

(1) 新建一个 ASP.NET MVC 2 空 Web 应用程序，命名为 MvcApp。

(2) 这里我们要处理员工信息列表，所以需要定义一个封装员工信息的实体类。员工信息一般有编号、姓名、性别、部门和职务等，所以这个员工实体类要定义几个属性，如下所示：

```
public class Worker
{
    public string Number { get; set; }
    public string Name { get; set; }
    public string Sex { get; set; }
    public string Department { get; set; }
    public string Post { get; set; }
}
```

(3) 创建一个人力资源模块的 Controller，命名为 HumanResourceController。

(4) 修改 HumanResourceController 下面的 Index 方法。在该方法里我们需要得到一个员工信息列表的集合，然后将该集合传递到视图中。代码如下：

```
public ActionResult Index()
{
    IList<Worker> workers = new List<Worker>();
    workers.Add(new Worker() {
        Number = "hr00132", Name = "张耀文", Sex = "男",
```

```
        Department = "开发部", Post = "软件工程师" });
    workers.Add(new Worker() {
        Number = "hr00133", Name = "郭楷", Sex = "男",
        Department = "工程部", Post = "实施工程师" });
    workers.Add(new Worker() {
        Number = "hr00134", Name = "柴兴", Sex = "男",
        Department = "开发部", Post = "程序员" });
    workers.Add(new Worker() {
        Number = "hr00136", Name = "李贝贝", Sex = "女",
        Department = "开发部", Post = "系统架构师" });
    workers.Add(new Worker() {
        Number = "hr00153", Name = "敬亮", Sex = "男",
        Department = "销售部", Post = "客户专员" });
    workers.Add(new Worker() {
        Number = "hr00156", Name = "刘飞", Sex = "男",
        Department = "销售部", Post = "客户经理" });
    workers.Add(new Worker() {
        Number = "hr00157", Name = "赵大志", Sex = "男",
        Department = "销售部", Post = "客户专员" });

    return View(workers);
}
```

(5) 在站点根目录下的 Views 目录下创建一个对应于 HumanResourceController 的子文件夹，命名为 HumanResource。

(6) 在 HumanResource 目录下创建一个默认视图，命名为 Index，得到一个 Index.aspx 的视图文件。

(7) 打开 Index.aspx 文件，修改页面@Page 指令，如下所示：

```
<%@Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<IList<MvcApp.Models.Worker>>" %>
```

(8) 在 Index 视图的 body 部分添加页面结构，获取传递过来的员工列表集合，然后遍历输出它们。页面主要代码如下：

```
<table class="tbl worker" cellspacing="1" cellpadding="5">
<thead>
<tr><td>编号</td><td>姓名</td><td>性别</td><td>部门</td><td>职务</td></tr>
</thead>
<tbody>
<% foreach (var worker in Model)
   { %>
<tr>
   <td><%= worker.Number %></td>
   <td><%= worker.Name %></td>
   <td><%= worker.Sex %></td>
   <td><%= worker.Department %></td>
   <td><%= worker.Post %></td>
</tr>
<%} %>
</tbody>
</table>
```

如此就完成了对数据集合的显示功能，下面我们来运行一下看看效果。

7.1.3 运行结果

运行该项目，访问/HumanResource/Index 路径，结果如图 7-1 所示。



图 7-1 员工信息列表

7.1.4 实例分析

**源码解析**

本实例首先创建了一个用于封装用户信息的实体类 Worker，然后在 HumanResource Controller 里的 Index 动作中创建一个存储员工列表信息的集合，并使用 View 方法将该集合以强类型的数据传递方式传递到视图中。在视图中接收该集合，使用 foreach 语句遍历该集合并取出集合中的对象，呈现到页面中。

7.2 为什么在 MVC 中可以使用 WebForm 控件

不得不承认，最初 MVC 并没有被微软的研发人员设计到 ASP.NET 中。至于具体为什么要在后来强加入 ASP.NET 中，我们无从得知。

不过在 Java 平台中的 Struts 之类的 MVC 应用框架如火如荼地发展，而且被很多开发人员给它们套上了一层层神圣的光环，微软如果没有推出一些相应的措施，确实有些丢软件帝国的面子。

视频教学：光盘/videos/07/ 7.2 为什么在 MVC 中可以使用 WebForm 控件 长度：8 分钟

7.2.1 软件帝国的超级武器——WebForm

WebForm 确实是一个很有创意、很实用的东西，不管是现在，还是在不远的将来，都将

一直保持着它独具的创意，以及它无与于伦比的方便性。

微软开创性地将桌面应用程序的开发模式引入对 Web 应用程序的开发中：拖控件、加事件处理程序，然后运行……这一切都是那么惬意，那么让人拍案叫绝。

而且，ASP.NET 将前台的展示代码和后台的处理程序完全隔离在两个不同的文件中，一切看似神奇的东西都被系统“自动”完成了。

这一切，对于初级程序开发人员来说，比让其写大堆的页面代码要舒服得多，所以 ASP.NET 一度吸引了无数的开发人员加入进来。

7.2.2　超级武器也有盲区

回到 20 世纪，大部分的 Web 应用都是使用页面中混合代码的数据编程，例如 ASP 和 PHP 直接在页面中向数据库请求并用 HTML 显示的程序结构。这种方式往往开发速度比较快，但是由于数据页面的分离不是很直接，所以很难体现出程序业务模型的样子，当然也很难实现模型的重用性。产品设计没有弹性，很难满足不断变化的用户需求。

不过这些缺陷都被 JSP 和 ASP.NET 这两种天生就面向对象的语言很好地解决掉了。现在一般的应用程序都使用三层(或多层)结构来处理程序中的业务逻辑。

在 Web 应用程序的界面层，经常需要在不同的时刻为不同的用户呈现不同的内容，所以能不能把这个对“不同的时刻”、“不同的用户”以及“不同的内容”的判断和具体的内容进行分离，使之更容易被控制和协调就成了一个新的问题。

于是，Java 随即引入了 MVC 模式的应用框架，很完美地解决了这个问题。

与此同时，ASP.NET 却在专注于 WebForm 的研究和应用，同时也取得了一些非常可观的成就。但是其极大程度地整合了 MVC 模式中的 V 和 C 的关系，虽然独具特色，却使应用程序的 UI 层耦合性加强，或者说有点与 MVC 的思想背道而驰(或许这么说有点贬义，不过这里的确没有贬低 WebForm 的意思)，惹来了不少开发和测试人员的抱怨。

7.2.3　软件帝国的快速反应

正值 Java 中的 MVC 框架快速发展的同时，ASP.NET WebForm 也大行其道，随即微软推出了 ASP.NET MVC 框架。

也许是因为 Java 中的 MVC 发展得太火了，微软有点跟风的意味，于是很多人开始见风使舵，大呼 ASP.NET 将迎来 WebForm 的末日、MVC 的新纪元，这样其实是毫无依据的。

首先，MVC 和 WebForm 根本不具有可比性，WebForm 是一个全新的 Web 开发模式，从头到尾都是一套完整的东西。虽然其在某些方面有一些局限性，但是仍然是一个优秀的开发模式，是 MVC 不可取代的。

其次，MVC 和 WebForm 各有所长，MVC 主要解决的是视图层的耦合性问题，但是其不具备快速开发和简单易用的特性，需要开发人员掌握的知识比较系统、全面，而 WebForm 却简单易用，入门非常容易，其具有 MVC 不可替代的位置。

总之，MVC 和 WebForm 不会冲突，也不会有哪一方很快消亡(除非有一天谁整合了它们二者的优点到某一模式上)。

就当下据微软对二者的态度来看，似乎应了我国的一位领导人所说过的一句话：两手抓，两手都要硬。



说一点可能让 Java 程序员心里不爽的话，学了 ASP.NET，感觉非常好(虽然它很“年轻”)，它直接解决掉了许多 Java 中的诟病，例如像长篇小说似的配置文件。

7.2.4 MVC 和 WebForm 的互补

看到 ASP.NET MVC，感觉像是一个轮回：似乎我们又被 MVC 带回到了“炸酱面”式代码的时代。服务器模型不能用，拖放的服务器控件不让用，事件驱动不能用……真的回到“侏罗纪”时代了吗？当然没有。

虽然 ASP.NET MVC 主张我们放弃 WebForm 中的很多习惯和思想，但绝对不会让我们回归“远古”的。就当是一个轮回(毕竟太像了，这点不得不承认)，也是在进化中的一个新时代，并不是那个远古时代。可以说，这是一次小范围的革命。

所有优秀的新事物的成长与发展的道路都是曲折的、多磨难的，但是其前途肯定是光明的。ASP.NET MVC 刚出道没几年，它还很年轻，有些不完美在所难免。但是我们可以使用 WebForm 的优势来弥补这些不足，例如下面要讲的迭代控件 Repeater 和 DataList 等。



ASP.NET MVC 开发小组极力反对这么做，但是确实可以这样解决一些问题，适用的就是最好的。

既然 ASP.NET MVC 和 ASP.NET WebForm 是不同的两个东西，那么如何在 MVC 的视图中使用 WebForm 中的控件呢？这个问题得归于 ASP.NET MVC 框架的实现方式。

默认的 ASP.NET MVC 使用 WebForm 中的视图引擎，所有 ASP.NET MVC 视图都将继承自 ViewPage 类。页面@Page 指令如下：

```
<%@Page Language="C#" Inherits="System.Web.Mvc.ViewPage<dynamic>" %>
```

而 ViewPage 类又继承自 Page 类。ViewPage 类的实现如下：

```
using System;
using System.IO;
using System.Web;
using System.Web.UI;

namespace System.Web.Mvc
{
    public class ViewPage : Page, IViewDataContainer
    {
        /* 类的实现代码略 */
    }
}
```

我们知道 Page 类在 ASP.NET WebForm 中将被所有的 Web 窗体继承，所以可以在 ASP.NET MVC 的视图页面中使用 WebForm 服务器端控件。



不能在视图中使用服务器端控件的时候使用控件的事件机制，但是官方不建议这么做，因为这样会使 MVC 视图变得不伦不类。

7.3　使用 Repeater 显示商品信息列表

前面我们演示了如何在 View 中使用 foreach 遍历数据集合，取出数据并展示到页面中。不难发现，我们又在大量地使用<% %>来标记一些服务器端代码。其实在 ASP.NET MVC 中的视图里同样可以使用 Web 服务器端控件，如此我们可以使用 Repeater 之类的控件来实现一些集合数据展示。

**视频教学：**光盘/videos/07/7.3　使用 Repeater 显示商品信息列表　　长度：11 分钟

7.3.1　基础知识

1. Repeater 控件

Repeater 控件是一个纯粹的集合数据迭代控件，它仅仅封装了页面上的遍历脚本，使用它不会在页面中生成任何 HTML 代码。它的用法非常简单，但是功能却十分强大。

Repeater 控件提供了 5 个数据模板，说明分别如下。

- HeaderTemplate　头部模板。因为数据列表一般会有表头，为保证代码结构化，建议将表头的内容放在这里。
- ItemTemplate　项目模板。这就是普通项的模板，也就是对数据列表里每一项在页面上展示的效果进行定义。
- AlternatingItemTemplate　交替项模板。对应 ItemTemplate，如果设置该项，该项表示偶数项的模板，一般设置列表奇偶项不同背景色时会用到。
- SeparatorTemplate　间隔符模板。在每一个 ItemTemplate 或 AlternatingItemTemplate 项之间插入分隔所用的内容。
- FooterTemplate　脚注模板。一般的列表项可能会用脚注说明该列表的信息，这里定义列表脚注的内容。

例如要在页面中显示下面的表格结构：

```
<table>
    <thead>
        <tr><td>标题</td></tr>
    </thead>
    <tbody>
        <tr><td>这是一行数据</td></tr>
        <tr><td>这是一行数据</td></tr>
        <tr><td>这是一行数据</td></tr>
    </tbody>
    <tfoot>
        <tr><td>脚注</td></tr>
    </tfoot>
</table>
```

可以使用 Repeater 控件来实现，代码如下：

```
<asp:Repeater ID="Repeater1" runat="server">
    <HeaderTemplate>
```

```
    <table>
        <thead>
            <tr><td>标题</td></tr>
        </thead>
        <tbody>
    </HeaderTemplate>
    <ItemTemplate>
            <tr><td>这是一行数据</td></tr>
    </ItemTemplate>
    <FooterTemplate>
        </tbody>
        <tfoot>
            <tr><td>脚注</td></tr>
        </tfoot>
    </table>
    </FooterTemplate>
</asp:Repeater>
```

当然，在该 Repeater 的数据源中还需要有一个数据集合。

2. 数据绑定方法

ASP.NET WebForm 中的数据迭代控件的数据绑定方法有 3 种，分为两类：一类是绑定当前数据元素，另一类是绑定数据元对象的属性。

第一类的绑定方法使用如下语句：

```
<%# Container.DataItem %>
```

这种方法可以绑定数据并直接打印到页面的对象集合中。例如基本数据类型的数组，或重写了 ToString()方法的对象集合等。

第二类方法使用 TemplateControl 类的 Eval()方法，其格式如下：

```
<%# Eval("ParameterName") %>
```

该方法将绑定一个当前数据对象的属性值，参数是一个字符串类型的对象，用于指定一个属性的名称。

另外，Eval()还可以添加一个字符串类型的字符串格式化参数，用于格式化输出时绑定的对象的属性值。

在代码中，我们可以使用 Repeater 对象的 DataSource 属性为其设置一个数据源。DataSource 属性可以接收一个数据集合或数据源对象作为数据源。

设置完数据源，就可以使用 DataBind()方法刷新该对象的页面结构，以重新绑定数据。

例如我们可以使用以下方法向页面的 Repeater1 对象绑定一个数据源并刷新 Repeater 对象：

```
this.Repeater1.DataSource = Model;
this.Repeater1.DataBind();
```

这里的 Model 是页面中的 Model 对象，它将接收使用强类型方式传递过来的数据集合。

7.3.2 实例描述

公司以前接过一个项目，要为一个做商品批发的贸易公司开发一个网站系统，其中不可缺少的就是商品管理功能。

本实例就以这个网站后台中的产品管理列表为例，展示一下 Repeater 对象在 MVC 视图中的用法。

7.3.3　实例应用

【例 7-2】使用 Repeater 显示商品信息列表。

(1) 打开前面我们使用的项目。

(2) 这里我们要处理商品信息，所以需要在 Models 目录下创建一个封装商品信息的实体类。商品信息一般有编号、名称、价格、类型和规格等信息，所以我们的商品实体类结构如下：

```
public class Product
{
    public string Number { get; set; }
    public string Name { get; set; }
    public string Price { get; set; }
    public string Type { get; set; }
    public string Standard { get; set; }
}
```

(3) 然后需要在 Controller 目录中创建一个处理产品操作请求的 Controller，命名为 ProductController。

(4) 我们需要以强类型方式向视图传递一个数据集合，所以这里修改 ProductController 中的 Index 方法，如下所示：

```
public ActionResult Index()
{
    IList<Product> list = this.GetProductList();
    return View(list);
}
```

这里还需要一个准备数据的 GetProductList()方法，其代码如下：

```
private IList<Product> GetProductList()
{
    IList<Product> list = new List<Product>();
    list.Add(new Product() { Number = "P001", Name = "统一矿泉水",
        Price = "￥1.5", Type = "副食品", Standard = "500ml" });
    list.Add(new Product() { Number = "P002", Name = "统一绿茶",
        Price = "￥2.5", Type = "副食品", Standard = "550ml" });
    list.Add(new Product() { Number = "P003", Name = "大骨面",
        Price = "￥1", Type = "副食品", Standard = "95g" });
    list.Add(new Product() { Number = "P004", Name = "周住牌洗衣粉",
        Price = "￥3.5", Type = "日化用品", Standard = "600g" });
    list.Add(new Product() { Number = "P005", Name = "GC 沐浴露",
        Price = "￥13", Type = "日化用品", Standard = "230ml" });
    list.Add(new Product() { Number = "P007", Name = "好老公晾衣架",
        Price = "￥75", Type = "家居用品", Standard = "200*60*120(cm)" });
    list.Add(new Product() { Number = "P008", Name = "热的快电饭锅",
        Price = "￥129", Type = "厨具", Standard = "2.5L" });
    return list;
}
```

(5) 在 Views 目录下创建一个名为 Product 的子文件夹。

(6) 在 Product 目录中添加一个视图文件，取名为 Index，新文件全名为 Index.aspx。

(7) 要接收强类型的集合数据，所以要修改 Index.aspx 文件中的@Page 指令，如下所示：

```
<%@Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<IList<MvcApp.Models.Product>>" %>
```

(8) 修改 Index.aspx 文件中的页面结构，添加一个 Repeater 控件，并编辑该控件的结构，为其绑定数据集合，主要代码如下：

```
<%
    this.rptProductList.DataSource = Model;
    this.rptProductList.DataBind();
%>
<asp:Repeater ID="rptProductList" runat="server">
<HeaderTemplate>
<table id="plist" border="0" cellpadding="0" cellspacing="0">
<thead>
<tr><td>产品编号</td><td>产品名称</td><td>产品价格</td><td>产品分类</td><td>产品
规格</td></tr>
</thead>
<tbody>
</HeaderTemplate>
<ItemTemplate>
<tr>
    <td><%# Eval("Number") %></td>
    <td><%# Eval("Name") %></td>
    <td><%# Eval("Price") %></td>
    <td><%# Eval("Type") %></td>
    <td><%# Eval("Standard") %></td>
</tr>
</ItemTemplate>
<FooterTemplate>
</tbody>
</table>
</FooterTemplate>
</asp:Repeater>
```

(9) 最后保存所有文件，然后可以运行查看效果了。

7.3.4　运行结果

运行项目，访问/Product/Index 路径，结果如图 7-2 所示。



图 7-2　产品列表页面

7.3.5　实例分析



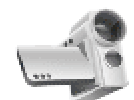
源码解析

本实例首先创建了一个用于封装商品信息的实体类 Product，然后在 ProductController 里的 GetProductList()方法中创建一个存储商品信息的集合，并在 Index 动作中使用该方法获取集合，接着使用 View 方法将该集合以强类型的数据传递方式传递到视图中。在视图中接收该集合，并将该集合绑定到 Repeater 控件。Repeater 控件将自动遍历该集合并取出集合中的对象，呈现到页面中。

7.4　使用 DataList 显示班级座位排列情况

Repeater 控件不会在页面中产生任何代码，所以我们可以使用 Repeater 随意控制页面代码。但是有时候我们需要生成固定列数的横排或竖排表格数据，使用 Repeater 显然有点不尽如人意。

本节我们将使用 DataList 在视图中生成一个指定行数的页面。



视频教学： 光盘/videos/07/7.4　使用 DataList 显示班级座位排列情况　　长度：6 分钟

7.4.1　基础知识

DataList 称为数据列表控件，和 Repeater 控件一样，也是迭代控件，它能够以事先指定的样式和模板重复显示数据源中的数据，不过它会默认在数据项目上添加表格来控制页面布局。DataList 控件集成了很强大的功能，并提供了多个模板及样式属性供我们使用。

DataList 控件支持 7 种模板，并为所有模板提供了相应的样式，在 MVC 视图中不常用到，这里就不做介绍了。

DataList 控件有两个重要的属性，如下所示。

- RepeatDirection　项目排列方向。值为 RepeaterDirection 枚举值，有 Horizontal(横向)和 Vertical(纵向)两项可以选择。
- RepeatColumns　重复列数。显而易见，就是生成的表格每行的列数。



DataList 控件的 RepeatColumns 属性有点意思。它指定的是列数，所以无论排列方向是横向还是纵向，它都将尽量生成这么多列(除非你的项数少于列数)。

DataList 控件绑定数据的方法和 Repeater 控件一样，这里就不多介绍了。

7.4.2　实例描述

某一天，有个当老师的朋友给我打电话说能不能帮他的学生管理系统中添加一个学生座位信息显示的页面，这当然是小菜一碟。

首先我了解他的学校一个班里最多也就 20 人，一排就 6 个学生，大概三四排。拿到需求

我的第一直觉就想到了 DataList 控件。

接下来我们就使用 DataList 控件为这个班里的学生排排座位。

7.4.3 实例应用

【例 7-3】使用 DataList 显示班级座位排列情况。

(1) 打开前面我们使用的项目。

(2) 这里我们要处理学生信息，所以需要在 Models 目录下创建一个封装学生信息的实体类。这里我们只需要关心学生的学号、姓名和性别，所以学生实体类结构如下：

```
public class Student
{
    public string Number { get; set; }
    public string Name { get; set; }
    public bool Sex { get; set; }
}
```

(3) 然后我们需要在 Controller 目录中创建一个处理学生操作请求的 Controller，命名为 StudentController。

(4) 需要以强类型方式向视图传递一个数据集合，所以这里修改 StudentController 中的 Index 方法，如下所示：

```
public ActionResult Index()
{
    IList<Student> list = this.GetStudentList();

    return View(list);
}
```

这里还需要一个准备数据的 GetStudentList()方法，代码如下：

```
public IList<Student> GetStudentList()
{
    IList<Student> list = new List<Student>();
    list.Add(new Student() { Number = "stu0201", Name = "王小宝", Sex = true });
    list.Add(new Student() { Number = "stu0212", Name = "李小贝", Sex = false });
    list.Add(new Student() { Number = "stu0202", Name = "张大柱", Sex = true });
    list.Add(new Student() { Number = "stu0203", Name = "刘小星", Sex = true });
    list.Add(new Student() { Number = "stu0210", Name = "李小冉", Sex = false });
    list.Add(new Student() { Number = "stu0204", Name = "周星星", Sex = true });
    list.Add(new Student() { Number = "stu0209", Name = "刘飞", Sex = true });
    list.Add(new Student() { Number = "stu0205", Name = "李三凤", Sex = false });
    list.Add(new Student() { Number = "stu0206", Name = "景玉", Sex = false });
    list.Add(new Student() { Number = "stu0207", Name = "马二立", Sex = true });
    list.Add(new Student() { Number = "stu0208", Name = "冯涛", Sex = true });
    list.Add(new Student() { Number = "stu0211", Name = "宋珊", Sex = false });
    list.Add(new Student() { Number = "stu0213", Name = "蔡丽", Sex = false });
    return list;
}
```

(5) 在 Views 目录下创建一个名为 Student 的子文件夹。

(6) 在 Student 目录中添加一个视图文件，取名为 Index，新文件全名为 Index.aspx。

(7) 我们要接收强类型的集合数据，所以这里修改 Index.aspx 文件中的@Page 指令，如下所示：

```
<%@Page Language="C#"
Inherits="System.Web.Mvc.ViewPage<IList<MvcApp.Models.Student>>" %>
```

(8) 修改 Index.aspx 文件中的页面结构，添加一个 DataList 控件，并编辑该控件的结构，为其绑定数据集合，主要代码如下：

```
<%
    this.dlStudentList.DataSource = Model;
    this.dlStudentList.DataBind();
%>
<asp:DataList ID="dlStudentList" runat="server" RepeatColumns="6"
RepeatDirection="Horizontal" >
<ItemTemplate>
<div class='<%# bool.Parse(Eval("Sex").ToString()) ? "stu item boy" :
"stu item girl"%>'>
    <div class="name"><%# Eval("Name") %></div>
    <div class="number"><%# Eval("Number") %></div>
</div>
</ItemTemplate>
</asp:DataList>
```

(9) 最后保存所有文件，可以运行查看效果了。

7.4.4　运行结果

运行项目，访问/Student/Index 路径，结果如图 7-3 所示。



图 7-3　学生座位排列情况

7.4.5　实例分析



源码解析

本实例首先创建了一个用于封装学生信息的实体类 Student，然后在 StudentController 里的 GetStudentList()方法中创建一个存储学生信息的数据集合，并在 Index 动作中使用该方法获取

集合，接下来使用 View 方法将该集合以强类型的数据传递方式传递到视图中。在视图中接收该集合，并将该集合绑定到 DataList 控件中。DataList 控件将自动遍历该集合并取出集合中的对象，呈现到页面中。

7.5 常见问题解答

7.5.1 在 MVC 中使用服务器端控件有什么规则



在 MVC 中使用服务器端控件有什么规则吗？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

学习了 ASP.NET MVC 2，觉得在 View 中使用服务器端控件有很大的便利，确定非常好用。不过不知道有没有禁忌，希望精通 MVC 的高手帮我解答这个问题。

【解决办法】答案是当然不能任意、自由地使用。

首先在 MVC 中又重新回到了表单提交的请求模式，而在 WebForm 中使用的是 PostBack 方式，虽然二者原理一致，但 WebForm 封装的东西太多，页面机制太复杂，开发人员不容易控制。而 MVC 就是为了解决这个问题，所以这里不建议使用和 WebForm 页面机制有过多牵扯的东西，例如回调。

其次，微软 MVC 开发小组的人员极力反对在 MVC 视图中使用服务器端控件，自然有他们的道理。或许，替代服务器端控件功能的特殊组件马上就要面世了。

不过，在这里我强调一点：千万不可以在使用服务器控件的时候触发控件的任何事件，否则你的 MVC 应用程序就真的被打入万劫不复之地了。

7.5.2 怎样实现 DropDownList 控件的 OnSelectedIndexChanged 事件



我怎么实现 DropDownList 控件的 OnSelectedIndexChanged 事件呢？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

本人刚接触 MVC，还有很多地方不明白。

我看了一下 MVC 方面的开发资料，觉得 MVC 利用 Controllers 将数据传送到 Views 中显示，然后在 Views 中利用 HTML 控件显示或操作数据(MVC 不建议使用服务器控件)。但本人觉得只利用 HTML 控件有时候很不方便。

例如使用服务器控件中的 DropDownList，它有个 OnSelectedIndexChanged 属性，当选择某一项时就可以通过 post_back 执行后台相应方法和返回相应结果在当前页面显示。而 MVC 只在 View 中利用类似于 Html.DropDownList()这样的方法怎样才能实现呢？是利用 Controllers 中的 Action 吗？而且要实现某些效果的事件，感觉利用服务器控件方便很多。

【解决办法】一分为二来说吧，在 WebForm 中使用服务器端控件确实有太多优点，甚至让人有点爱不释手。但是从 MVC 的角度来看它却不是最好的，因为它的耦合度太高。

在 MVC 中，所有的浏览器请求都需要由 Controller 接收并处理，所以 WebForm 中的 PostBack 机制显得有点格格不入。

至于你说的当下拉列表框选中某值的时候进行一些处理，你可以在下拉列表框的 On SelectChange 事件处理程序中使用 Ajax，异步请求 Web 服务器，然后获取请求结果并进行相应的操作就可以了。

可能相对于服务器端控件的事件机制有点费事，但也是不得已而为之。希望哪一天微软能把 MVC 和 WebForm 的优势集成到一起吧。

7.6　习　　题

一、填空题

(1) 在 ASP.NET MVC 中，ViewPage 类继承自__________类。

(2) 在 DataList 控件的所有属性中，__________属性将用来指定数据项显示的列数。

(3) 下面代码是为页面中一个 Repeater 控件绑定数据源，请补充完下面代码。

```
this.Repeater1.DataSource = Model;
this.Repeater1.__________;
```

(4) 在 Repeater 中，如果要实现交替项效果，则需要使用到__________模板。

二、选择题

(1) 在 Repeater 控件的所有属性中，__________属性可以为 Repeater 控件指定一个数据源。

A. Data　　B. Source

C. Model　　D. DataSource

(2) 使用 DataList 控件的 RepeatDirection 属性，可以设置该控件中数据项排列的方向，其可选项有__________和 Vertical。

A. Right　　B. Down

C. Horizontal　　D. Row

(3) 在 ASP.NET MVC 中，__________方法可以在 Repeater 的模板项中绑定当前数据对象的 Name 属性值。

A. `<%# Eval("Name") %>`

B. `<%# Value("Name") %>`

C. `<%= Eval("Name") %>`

D. `<%= Container.DataItem %>`

三、上机练习

上机练习：使用 Repeater 控件显示个人通讯录。

每个人都会有很多需要联系的人，朋友、客户、同事、同学和家人，总体数量往往上百。如果把所有的信息都记录在手机上显然是不可能的，而使用一个电话本来记录，不容易查询，又容易丢失，所以某企业在做 OA 系统的时候提出了做个人通讯录的需求。

系统可以让个人记录并查询自己的通讯录信息。通讯录中需要记录联系人的姓名、关系、电话号码、手机号码、住址以及其备注信息。

在这次练习中，我们需要使用服务器端控件 Repeater 来在视图中呈现个人通讯录的列表，如图 7-4 所示。



姓名	关系	电话	手机	地址
张三林	朋友		13636363636	河南省新郑市
李贝贝	高中同学	0371-62446666	13789123456	河南省新郑市
王森	大学同学		15923578954	河南省郑州市
杨明	老师		18898654321	北京市
李刚	领导		15245678945	河北省

图 7-4　个人通讯录列表



第 8 章　自定义视图引擎

内容摘要

前面提到过，ASP.NET MVC 默认使用的是 WebForm 充当的视图引擎。或许是这个视图引擎还不太成熟，或许因为它基于 WebForm 而对它扩展性能有局限，现在它还不太好用。例如前面我们破格使用的 WebForm 迭代控件，就是被逼无奈的做法。总之，ASP.NET MVC 的视图引擎还不能称得上“优秀”。

不过，Microsoft 已经对 ASP.NET MVC 框架开放源代码，所以我们可以对它进行研究，根据它提供的一些接口开发适合自己的视图引擎。

本章我们就来了解一下 ASP.NET MVC 生成视图的原理，并且使用 MVC 框架为提供的接口来实现一个简单的自定义视图引擎。

学习目标

- 了解 ASP.NET MVC 应用程序生成视图的原理
- 熟练使用模板引擎 StringTemplate
- 掌握自定义视图引擎的方法

8.1 使用代码拼凑的简单登录页面

既然要自定义视图引擎，当然就不能使用 ASP.NET MVC 自带的 WebForm 视图引擎了。前面我们学过不用视图也可以向页面响应数据，其实 ASP.NET MVC 中的视图引擎就是基于此向客户端响应数据的。本节我们来重温一下使用最基本的代码拼凑一个页面的方法，深入了解 ASP.NET 的视图引擎机制。

**视频教学：**光盘/videos/08/8.1 使用代码拼凑的简单登录页面 长度：10 分钟

8.1.1 基础知识——视图生成的原理

要想了解视图引擎怎么工作，必须先知道视图引擎要干什么。

1. 视图的本质

视图是响应给浏览器的一些 HTML 代码，也就是在浏览器中查看源文件时所看到的内容，也叫 Web 页面。

查阅 Web 的历史可以知道，最早的 Web 就是用于在互联网上(可能当时还不叫互联网)共享信息，可以供远程计算机直接打开查看。当时的信息只有文本，也就是一些纯文字的东西。当时需要每个人知道文件具体的物理路径，也就是 URL(当时可能也不叫 URL，可能是其他名字)，通过 URL 可以直接查看远端计算机中的文件。

后来，随着 Web 的发展，加入了超链接、多媒体(图像、声音、视频和动画等)，还有一些样式、布局之类的东西，这些东西使用一些特殊的标签来表示，也就是所谓的 HTML(Hyper Text Mark-up Language，超文本标记语言)，这些标记通过 CSS 样式等辅助技术来控制页面的布局和样式等。

但是，无论 Web 怎么发展，它都是一些纯文本的东西，只是浏览器在处理这些文本的时候根据标记导入一些图片和动画等资源。也就是说，无论超文本标记语言如何“超”，它都没有改变文本的本质。在所有的 Web 服务器中，执行处理的主要信息以及返回的主要结果还是文本。



“Web 服务器中处理的主要内容是文本”这一核心思想要根深蒂固牢记在心，否则一些新手就可能被华丽的页面效果所迷惑。

Web 服务器的作用就是准备好这些 HTML 文本和一些资源文件(例如 CSS、JS、图像和声音等文件)等待浏览器请求。

例如，当用户在浏览器中输入一个 URL 请求某个 Web 页面的时候，Web 服务器接收到浏览器请求并根据用户请求的 URL 以及参数生成 Web 页面，然后响应给浏览器。浏览器在接收到服务器响应的 Web 页面(HTML 文本)的时候，解析这个 Web 页面，根据 HTML 中相应的标记去获取资源，并根据 CSS 中的内容布局页面结构，显示到页面中。执行流程如图 8-1 所示。

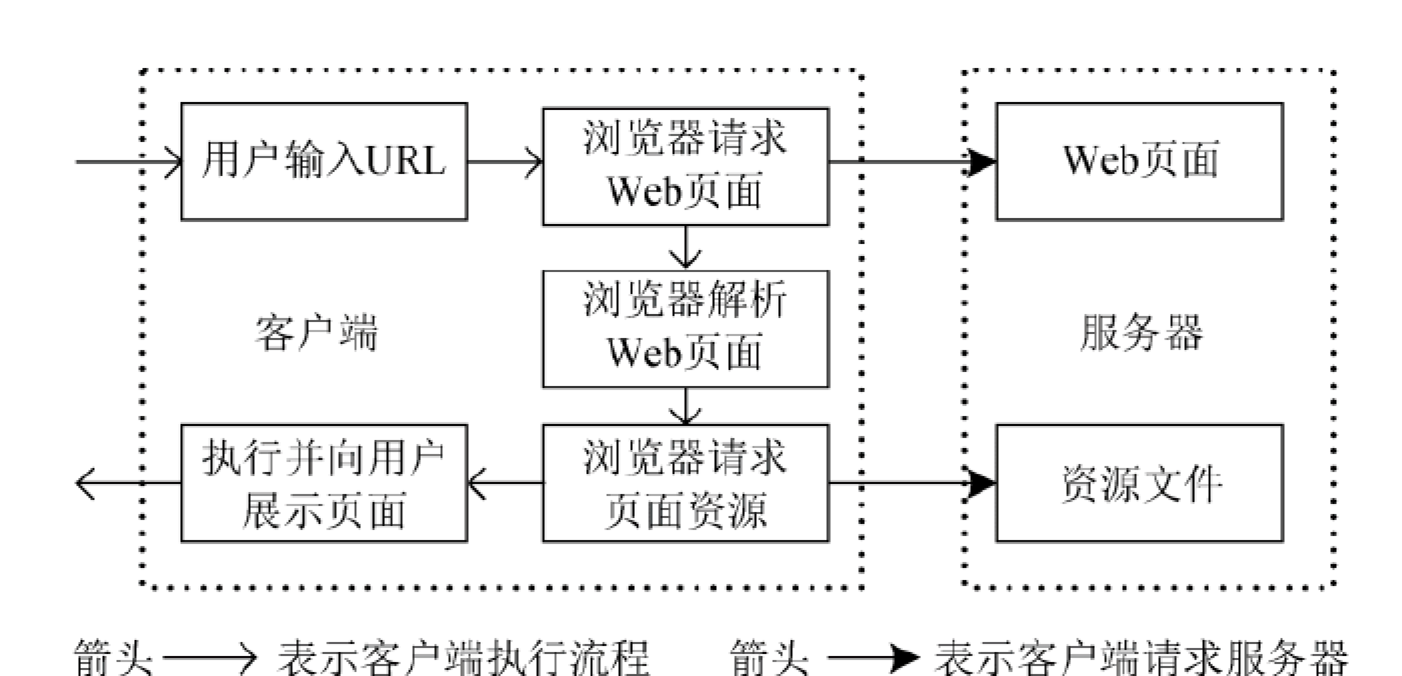


图 8-1　Web 请求执行流程

这里第一次客户端对服务器请求的是 Web 页面，主要是获取 Web 服务器动态生成的页面主体。在第二次及以后的请求中，才向 Web 服务器请求 CSS 文件、JS 文件、图像、声音以及动画等其他页面中关联到的资源文件。而我们讲的视图，就是在这个流程图中第一次请求 Web 服务器的时候所生成的那些 HTML 代码。



上面讲的是通用的 Web 应用程序的执行流程，不管是 ASP.NET MVC 还是 WebForm，或者其他如(PHP、JSP 之类)的动态语言，其主要功能都是动态生成 Web 页面。

2．视图的生成

前面讲过动态 Web 语言的主要功能都是生成 Web 页面，当然这里的 ASP.NET MVC 也不例外。

在 ASP.NET MVC 中，使用 Controller 接收用户请求。在 Controller 接收到用户请求的时候，其主要目的就是组织一段像下面代码结构一样的 HTML 文本。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>无标题文档</title>
</head>
    …
<body>
</body>
</html>
```

然后将这段文本响应给浏览器。浏览器接收到这段文本以后进行解析，就可得知如何向用户展示页面。

组织上面的那段代码并不复杂，前面我们也在 Controller 中直接使用 Response.Write()方法向响应流中写这么一段文本，其实视图引擎的内部实现就是如此。

在 ASP.NET MVC 中，视图引擎的作用就是获取相应的视图文件的内容，处理并填充相应的数据，得到最终的 HTML 代码，再写到响应流中。

8.1.2　实例描述

其实前面我们已经讲过在 Action 中使用 Response.Write()方法向响应流中写一段 HTML 文

本。为了加深印象，我们在这个实例中还使用 Response.Write()方法向浏览器响应数据。

本实例我们就向页面中输出一个非常简单的登录页面，不过需要改进一点，即在 HTML 文档中将需要使用的数据用数据替换的方式动态加入。

8.1.3 实例应用

【例 8-1】使用代码拼凑的简单登录页面。

(1) 新建一个空的 ASP.NET MVC 项目。

(2) 添加一个 Controller，命名为 LoginController。

(3) 在 LoginController 中添加一个私有方法 GetPageContent()，该方法将组织一个页面结构，代码如下：

```
/// <summary>
/// 获取页面内容
/// </summary>
private string GetPageContent()
{
   return "<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Transitional//EN'
'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>\n" +
       "<html xmlns='http://www.w3.org/1999/xhtml'>\n" +
       "   <head>\n" +
       "      <meta http-equiv='Content-Type' content='text/html;
charset=utf-8' />\n" +
       "      <title>$Title$</title>\n" +
       "   </head>\n" +
       "   <body>\n" +
       "   <form action='$PostAddress$' method='post'>\n" +
       "      <input name='username' value='$DefaultName$' ><br>\n" +
       "      <input name='password' ><br>\n" +
       "      <button type='submit' >登录</button>\n" +
       "   </form>\n" +
       "   </body>\n" +
       "</html>\n";
}
```

这段代码用字符串组织了一个 HTML 页面，不过在页面代码的 head 部分的 title 和 form 标记中的 action 属性以及 input 标记中的 value 属性分别使用两个美元符号括起的变量名称作为占位符，以备数据替换。



上面代码在每一个字符串后面都加入一个\n 符号，其目的是在页面中打印一个换行符，格式化页面代码。

(4) 再添加一个使用 ViewData 数据替换页面内容中占位符的方法，名为 ReplacePlaceHolder，代码如下：

```
/// <summary>
/// 替换占位符
/// </summary>
private string ReplacePlaceHolder(string pageContent)
{
```

```
    //遍历 ViewData，使用每一个数据元素的 Value 的值替换 Key 所标示的占位符
    foreach (var vd in ViewData)
    {
        string key = "$" + vd.Key + "$";
        pageContent = pageContent.Replace(key, vd.Value.ToString());
    }
    return pageContent;
}
```

该方法遍历 ViewData 中的项目，将 Key 作为替换的关键字，替换成相对应的值的内容。

(5) 修改 Index 动作，添加相应的代码，处理该次请求，代码如下：

```
public void Index()
{
    ViewData["Title"] = "登录页面";
    ViewData["DefaultName"] = "Admin";
    ViewData["PostAddress"] = "/Login/Post";

    //获取页面内容
    string pageContent = this.GetPageContent();

    //替换占位符
    pageContent = this.ReplacePlaceHolder(pageContent);
    //打印页面内容
    Response.Write(pageContent);
}
```

8.1.4　运行结果

运行项目，访问/Login/Index 路径，结果如图 8-2 所示。

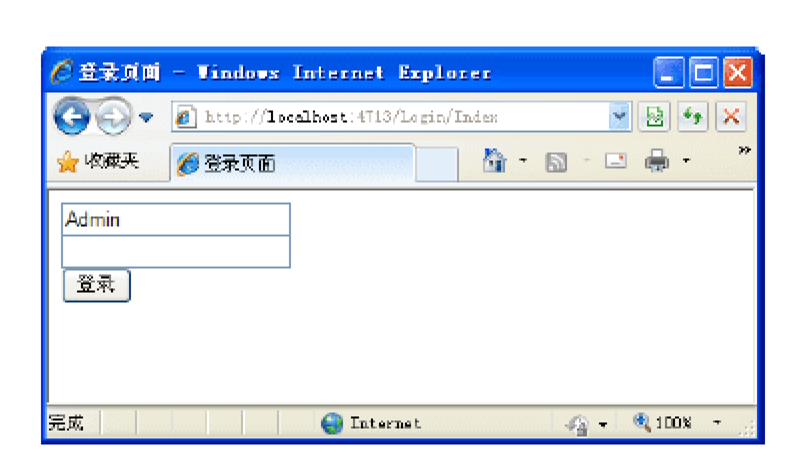


图 8-2　执行结果

在页面空白处右击鼠标，选择【查看源文件】命令，打开一个窗口。在该窗口中我们可以查看页面的源代码，也就是在服务器端生成的 HTML 代码。正常情况下我们看到的结果如下所示：

```
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Transitional//EN'
'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>
<html xmlns='http://www.w3.org/1999/xhtml'>
   <head>
       <meta http-equiv='Content-Type' content='text/html; charset=utf-8' />
       <title>登录页面</title>
   </head>
   <body>
   <form action='/Login/Post' method='post'>
```

```
        <input name='username' value='Admin' ><br>
        <input name='password' ><br>
        <button type='submit' >登录</button>
    </form>
    </body>
</html>
```

从上面代码可以看到，我们准备的 HTML 代码中的占位符都已经被替换成相应的内容。

8.1.5 实例分析



源码解析

本实例先在 Action 中存放 3 个对象到 ViewData 中，以备后面替换占位符使用。接着获得一个 HTML 页面内容，然后遍历 ViewData 字典，取出所有数据，使用每个数据项的值替换对应数据项的键的占位符，得到最终的页面内容。最后使用 Response.Write()方法将最终生成的内容写到响应流中。这样在浏览器访问的时候就可以看到替换过的页面内容了。

8.2 自定义视图引擎显示页面脚注信息

前面我们讲的是在程序中组织一段 HTML 代码。在实际应用中，没有人会把大段的 HTML 代码直接写到程序中。

其他缺点不用说，单从 MVC 的角度出发，这样做耦合性太强了，肯定不会有人这么做。如果你想尝试，保证你会很快放弃自定义视图引擎的学习。

对于这个问题，我们可以做一个非常简单的处理：把视图存到文本文件中，在使用的时候用 IO 技术将它们读取出来，进行操作。

下面我们来了解一下。



视频教学：光盘/videos/08/8.2　自定义视图引擎显示页面脚注信息　　长度：9 分钟

8.2.1 实例描述

首页的脚注部分有一些附加信息，例如版权、备案号、联系方式和公共安全备案号等内容。这些项一般都需要在后台进行修改，本节我们就使用自定义视图来将它们输出。

8.2.2 实例应用

【例 8-2】自定义视图引擎显示页面脚注信息。

(1) 打开上一节我们创建的项目。

(2) 我们需要在页面中显示脚注信息，所以在这里要在 Models 目录中创建一个封装信息的

实体类，命名为 ItzcnFooter，代码如下：

```
public class ItzcnFooter
{
    public string Copyright { get; set; }
    public string RecordNumber { get; set; }
    public string ContectWay { get; set; }
    public string PublicSecurityNumber { get; set; }
}
```

(3) 创建一个处理视图的类，并将它创建在 Models 目录中，命名为 ItzcnViewEngine，代码如下：

```
public class ItzcnViewEngine
{
    public static void View(ControllerContext context)
    {
        //获取视图内容
        string viewContent = GetViewContent(context);

        //替换占位符
        string html = ReplacePlaceHolder(viewContent,
context.Controller.ViewData);

        //输出页面内容
        context.HttpContext.Response.Write(html);
    }

    /// <summary>
    /// 替换占位符
    /// </summary>
    private static string ReplacePlaceHolder(string viewContent,
ViewDataDictionary viewData)
    {
        //遍历 ViewData，使用每一个数据元素的 Value 值替换 Key 所标识的占位符
        foreach (var vd in viewData)
        {
            string key = "$" + vd.Key + "$";
            viewContent = viewContent.Replace(key, vd.Value.ToString());
        }
        return viewContent;
    }

    /// <summary>
    /// 获取视图内容
    /// </summary>
    private static string GetViewContent(ControllerContext context)
    {
        //获得一个视图路径
        string viewPath = GetViewPath(context);

        //返回所有的文件内容
        return File.ReadAllText(viewPath);
    }

    /// <summary>
```

```
    /// 获取视图路径
    /// </summary>
    private static string GetViewPath(ControllerContext context)
    {
        HttpServerUtilityBase Server = context.HttpContext.Server;

        //得到当前执行的控制器和动作的名称
        string controller = context.RouteData.Values["controller"].ToString();
        string action = context.RouteData.Values["action"].ToString();

        //使用这组名称组成一个视图文件的虚拟路径
        string viewPath = "~/Views/" + controller + "/" + action + ".view";

        //返回该视图文件的物理路径
        return Server.MapPath(viewPath);
    }
}
```

该类提供了一个公有静态方法，用于向页面打印视图。其中 GetViewPath(ControllerContext context) 方法用于根据当前请求的动作名称获取一个视图文件的路径，GetViewContent(ControllerContext context)方法用于根据视图文件的路径获取文件的文本内容，ReplacePlaceHolder(string viewContent, ViewDataDictionary viewData)方法将使用 ViewData 中的数据替换视图文件中的占位符。而公有方法 View(ControllerContext context)则根据控制器的上下文调用另外 3 个方法来执行页面视图输出的功能。

(4) 创建一个 Controller，命名为 ItzcnController，并修改该 Controller 的代码，如下所示：

```
public class ItzcnController : Controller
{
    public void Index()
    {
        //获取页面脚注信息
        ItzcnFooter itzcnFooter = this.GetFooterInfo();

        //设置 ViewData
        ViewData["Copyright"] = itzcnFooter.Copyright;
        ViewData["RecordNumber"] = itzcnFooter.RecordNumber;
        ViewData["ContectWay"] = itzcnFooter.ContectWay;
        ViewData["PublicSecurityNumber"] = itzcnFooter.PublicSecurityNumber;

        //显示视图
        ItzcnViewEngine.View(ControllerContext);
    }

    /// <summary>
    /// 获取一个页面脚注信息
    /// </summary>
    private ItzcnFooter GetFooterInfo()
    {
        return new ItzcnFooter()
        {
            Copyright = "2005-2010 窗内网(www.itzcn.com)",
            RecordNumber = "豫 ICP 备 08104500 号",
            ContectWay =
```

```
                "在线客服 QQ 群 1: 33925615(已满)  QQ 群 2: 45368980(已满)" +
                "  QQ 群 3: 107423140(已满)  QQ 群:7858178",
            PublicSecurityNumber = "41010329000027"
        };
    }
}
```

在动作 Index 中，我们需要一组页面脚注信息，这些信息一般是从数据库中读取，所以这里使用一个方法来封装所获取数据的代码，模拟一组数据。在 Index 动作中接收到该对象，并填充到 ViewData 对象中，然后使用 ItzcnViewEngine 类的 View()方法调用视图。

(5) 当然，我们还需要一个视图文件。因为在 ItzcnViewEngine 类中，我们定义的文件路径规则是在站点目录下的 Controller 名称目录下以 Action 名称和.view 扩展名的视图文件，所以要在/Views/Itzcn 目录中创建一个名为 Index.view 的文件，并输入一个页面结构，其代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
    <style type="text/css">
        div{ font-size:12px; text-align:center; color:#143c80; font-family:宋
体; padding:8px; }
        #body{ width:766px; margin:auto; }
    </style>
</head>
<body>
<div id="body">
    <img alt="" src="/Images/itzcn.jpg" />
    <div>Copyrights Reserved $Copyright$  $RecordNumber$</div>
    <div>$ContectWay$</div>
    <div><img alt="" src="/Images/jt.gif" /></div>
    <div>郑公备: $PublicSecurityNumber$</div>
</div>
</body>
</html>
```

这里使用$符号标记了几个占位符：$Copyright$、$RecordNumber$、$ContectWay$和$PublicSecurityNumber$，分别用于表示版权信息、备案号、联系方式和公共安全号码等，这些占位符将在生成视图的时候被替换成相应的内容。

不过这里要注意，为了节省视图代码，页面的主体部分使用一个截图代替，这里直接使用一个 img 标签引入图片/Images/itzcn.jpg。

8.2.3　运行结果

运行应用程序，访问/Itzcn/Index 路径，结果如图 8-3 所示。

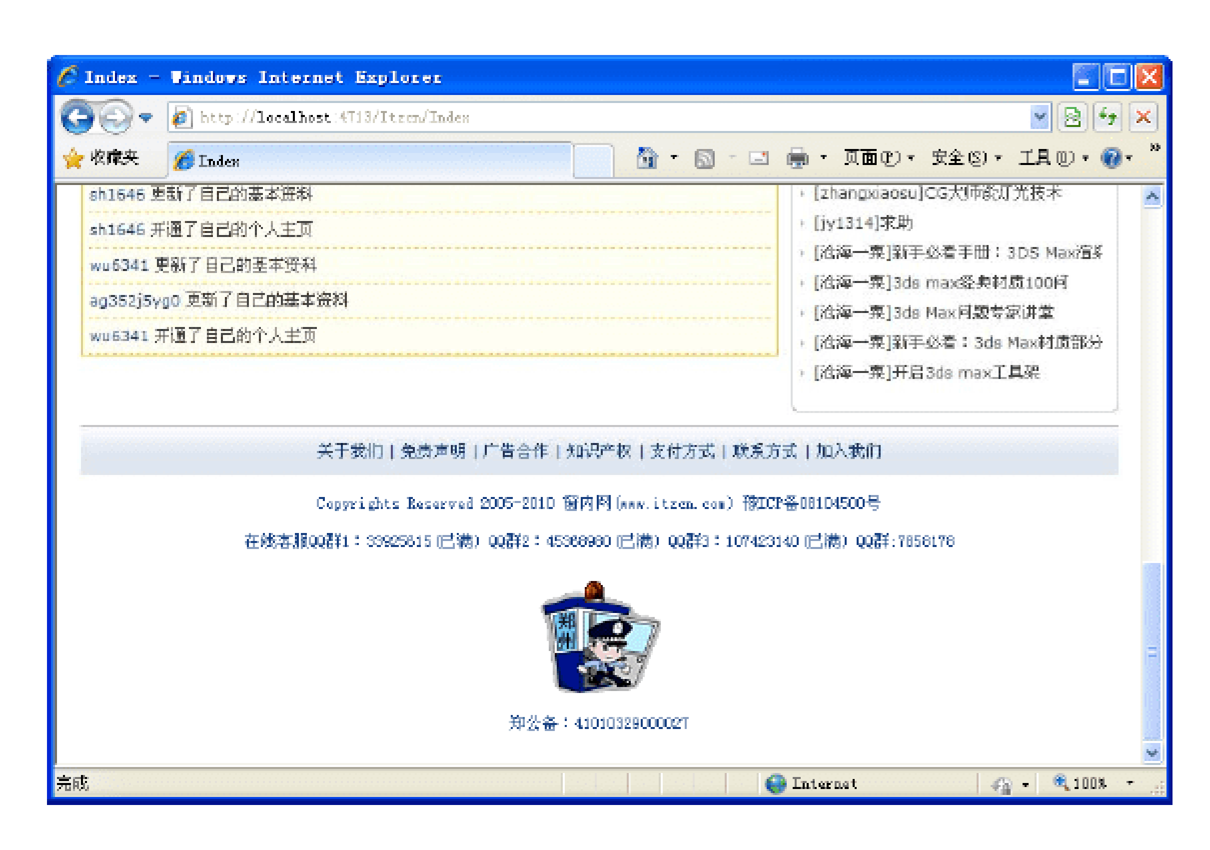


图 8-3　执行结果

可以看到，ASP.NET MVC 使用我们自定义的视图引擎输出了我们想要的内容。

8.2.4　实例分析



源码解析

本实例修改了 ItzcnController 的 Index 动作为无返回值动作，然后像平常一样使用 ViewData 保存页面的信息，最后调用 ItzcnViewEngine 类的 View()方法向响应流中输出页面内容。在 ItzcnViewEngine 类中，先根据当前 Controller 及 Action 的名称和相应的规则得到一个文件路径，然后打开并读取该文件的内容，并使用前面在 ViewData 中保存的数据替换视图文件内容中的占位符，最后使用 Response 对象的 Write()方法将 HTML 数据写到响应流中。

8.3　引入一个模板引擎优化自定义的视图引擎

上一节自定义的视图引擎已经比较完善了，但是还有一些小的问题，例如在进行页面占位符替换的时候，只能替换简单的字符内容，没有办法进行比较智能的替换操作。如果要实现更加复杂的操作，还需要更多的代码来完成这个功能。

对于这个问题，其实已经有很多非常好的解决方案了。下面我们将使用一个模板引擎 StringTemplate。



视频教学：光盘/videos/08/8.3　引入一个模板引擎优化自定义的视图引擎　长度：7 分钟

8.3.1　基础知识——StringTemplate 模板引擎

1．什么是 StringTemplate

StringTemplate 是一个非常好用的模板引擎，它的官方网站是 http://www.stringtemplate.org。

其官方网站上很清楚地说明了它是一个 Java 模板引擎(同时还有应用于 C#、Python、Ruby 和 Scala 等语言的版本)，它可以生成源代码、Web 页面、E-mail 和其他格式的文本输出。

因为 StringTemplate 要考虑多个平台使用方法的统一性，所以其从功能上来说称不上强大。但是其非常好用的特性不可小视，所以有不少开发者选择使用它。

2. StringTemplate 的使用

StringTemplate 模板引擎的当前最新版本是 3.2，我们可以从官方网站上找到相应的下载链接，下载得到应用于 C#语言的 StringTemplate 模板引擎的类库。

StringTemplate 模板引擎主要使用类 StringTemplate，该类声明在 Antlr3.ST 命名空间中，这个类基本上可以满足我们的需求。

首先我们要在项目中添加对该类库 Antlr3.StringTemplate.dll 的引用，应用程序会自动添加与之关联的其他类库。

在应用程序中，我们可以直接使用其重载的构造方法来声明该类的对象，例如下面的代码就可以创建一个 StringTemplate 类的对象。

```
var st = new StringTemplate(string template);
```

该构造方法的唯一参数 template 是一个视图的源代码。

接下来可以使用该类提供的 SetAttribute(string name, object value)方法来设置要替换的对象列表，使用它们可以很简单地替换视图源码中的占位符。

例如视图中源代码有下面一句文本：

```
<div>$UserName$</div>
```

使用 SetAttribute(string name, object value)方法可以将其完全替换，代码如下：

```
st.SetAttrubute("UserName", "张三凤");
```

上面代码执行以后，StringTemplate 对象可以生成如下代码：

```
<div>张三凤</div>
```

如果我们想要获取生成的结果，直接使用 StringTemplate 类对象的 ToString()方法即可。

看到这里，也许有的人已经发现，StringTemplate 仅仅为我们节省了很有限的一些代码，我们有什么必要使用它呢？

我们来继续研究它。

3. StringTemplate 的特色

前面我们使用一个字符串直接替换占位符。如果是一个对象，我们还要手动遍历，进行相应的替换，非常麻烦。

例如上一节实例中使用的 ItzcnFooter 类的实例，我们需要将它的属性一个个地添加到 ViewData 中，然后再进行遍历。

如果是一个非常复杂的对象，将会浪费很多不必要的代码。如果使用 StringTemplate 模板，我们只需要将这个实例对象设置到 StringTemplate 对象中即可，在视图中我们可以使用点运算符直接标识相应类的指定属性即可。

例如在控制器中执行以下代码：

```
var user = new { Name="张三凤", Age=23 };
st.SetAttribute("User", user);
```

在视图中就可以直接使用$User.Name$取得字符串“张三凤”，再使用$User.Age$取得年龄 23。



当然，StringTemplate 还有许多非常好用的特性，这里就不再一一介绍了，有兴趣的朋友自己研究一下吧。

8.3.2 实例描述

本节的内容是要引入 StringTemplate 模板引擎来优化我们的代码，本实例在上一节课的基础上进行修改，完善我们的自定义视图引擎。

8.3.3 实例应用

【例 8-3】引入一个模板引擎优化自定义的视图引擎。

(1) 打开上一节我们使用的项目。

(2) 首先修改 ItzcnController 中的 Index 动作，代码如下：

```
public void Index()
{
    //获取页面脚注信息
    ItzcnFooter itzcnFooter = this.GetFooterInfo();

    //设置 ViewData
    ViewData["Footer"] = itzcnFooter;

    //显示视图
    ItzcnViewEngine.View(ControllerContext);
}
```

可以看到这里的代码变得清爽了许多。

(3) 打开 ItzcnViewEngine 类的源代码，先在文件顶部添加对类库的 StringTemplate 类库的引用，代码如下：

```
using Antlr3.ST;
```

(4) 修改替换占位符的 ReplacePlaceHolder()方法，代码如下：

```
/// <summary>
/// 替换占位符
/// </summary>
private static string ReplacePlaceHolder(string viewContent,
ViewDataDictionary viewData)
{
    var st = new StringTemplate(viewContent);

    //遍历 ViewData，使用每一个数据元素的 Value 值来替换 Key 所标识的占位符
    foreach (var vd in viewData)
    {
```

```
        st.SetAttribute(vd.Key, vd.Value);
    }
    return st.ToString();
}
```

(5) 修改视图文件的代码，代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Index</title>
    <style type="text/css">
        div{ font-size:12px; text-align:center; color:#143c80; font-family:宋
体; padding:8px; }
        #body{ width:766px; margin:auto; }
    </style>
</head>
<body>
<div id="body">
    <img alt="" src="/Images/itzcn.jpg" />
    <div>Copyrights Reserved $Footer.Copyright$  $Footer.RecordNumber$</div>
    <div>$Footer.ContectWay$</div>
    <div><img alt="" src="/Images/jt.gif" /></div>
    <div>郑公备：$Footer.PublicSecurityNumber$</div>
</div>
</body>
</html>
```

这里将所有的占位符修改为指定以点运算符来选择属性的方式，更加易读，容易理解。

至此就算修改完了。比较简单，运行以后的结果和上一节完全相同，所以这里就不再截图显示其运行结果了。

8.3.4　实例分析



源码解析

本实例使用了 StringTemplate 模板引擎来优化我们的程序代码。主要实现了占位符替换环节的功能强化和扩展。

首先在向 ViewData 中保存数据的时候可以直接保存数据对象，然后在替换的时候直接设置对应的键和数据对象即可。StringTemplate 对象将自动遍历数据对象，取出所有的属性与页面中的相应位置进行替换，最后使用 StringTemplate 对象的 ToString()方法直接返回执行结果。

8.4　博客文章页面

前面几节自定义了一个视图引擎，似乎已经很完美了。事实上是这样吗？答案是 No。

虽然说前面的“视图引擎”已经实现了显示视图的功能，以及将视图与控制器相分离的功能，但是从严格意义上来说却不能称之为“视图引擎”，充其量可以叫做封装了 Response.Write()

方法的“视图生成器”而已。

为什么呢？很简单。“引擎”是一个系统中最核心的功能模块，而不仅仅是一个被动调用静态方法。

可能理由有点不沾边，也可能是我理解不深吧，只能如此解释了。不过当每个人看完前面几节后都会感觉有点别扭……废话不说，进入正题，本节我们使用系统提供的接口来实现一个真正意义上的视图引擎。

 **视频教学：** 光盘/videos/08/8.4　博客文章页面　　长度：15 分钟

8.4.1　基础知识——构建真正意义上的视图引擎

前面我们做的那个视图引擎的作用是根据 ControllerContext 对象的数据，输出页面视图的内容。

在 ASP.NET MVC 2 中，视图引擎的职责是根据参数选择并创建视图对象，视图引擎不负责视图内容的生成工作，视图内容的生成工作由视图对象来完成。

默认情况下，创建自定义视图引擎需要实现 IViewEngine 接口(该接口声明在 System.Web.Mvc 命名空间中)。该接口声明了 3 个方法，具体代码如下：

```
public interface IViewEngine
{
    ViewEngineResult FindPartialView(
        ControllerContext controllerContext,
        string partialViewName,
        bool useCache);

    ViewEngineResult FindView(
        ControllerContext controllerContext,
        string viewName,
        string masterName,
        bool useCache);

    void ReleaseView(
        ControllerContext controllerContext,
        IView view);
}
```

一般来说，几乎所有的模板引擎都会使用虚拟路径进行对视图文件的定位，并加载视图。如果我们使用 IViewEngine，还需要编写相应的路径处理代码。

为了方便，ASP.NET MVC 2 提供了一个基类 VirtualPathProviderViewEngine，并封装了一些路径处理程序。

VirtualPathProviderViewEngine 类是一个抽象类。该类包含了各种路径的查询模板，可以根据 controller 和 action 的值进行查询查应的视图文件。

另外 VirtualPathProviderViewEngine 类还包含两个用于创建视图的方法，这两个方法是抽象方法，需要我们在继承该类的时候实现这两个方法。

使用 VirtualPathProviderViewEngine 类还有一个好处：它会自动在生产环境下进行缓存。我们知道视图文件是在硬盘上存放的，而进行访问视图文件的 IO 操作很费资源。使用

VirtualPathProviderViewEngine 类就很简单地解决了这个问题。

总体来说，VirtualPathProviderViewEngine 类需要我们关心的类结构声明如下：

```
public abstract class VirtualPathProviderViewEngine : IViewEngine
{
    public string[] MasterLocationFormats { get; set; }
    public string[] PartialViewLocationFormats { get; set; }
    public string[] ViewLocationFormats { get; set; }

    protected abstract IView CreatePartialView(
        ControllerContext controllerContext,
        string partialPath);

    protected abstract IView CreateView(
        ControllerContext controllerContext,
        string viewPath,
        string masterPath);
}
```

也就是说，继承 VirtualPathProviderViewEngine 类需要我们提供各种视图文件路径的查询模板，以及创建视图的方法。在视图文件路径模板中，我们可以使用{0}、{1}的方式分别为 Action 和 Controller 名称提供占位符。

从上面 VirtualPathProviderViewEngine 类的结构我们可以看到，这两个抽象方法需要返回一个实现了 IView 接口的类的对象，所以这里还要完成一个实现 IView 接口的自定义视图对象类。

IView 接口的声明如下：

```
public interface IView
{
    void Render(ViewContext viewContext, TextWriter writer);
}
```

这里我们只要实现一个 Render()方法即可。

Render()方法需要两个参数 viewContext 和 writer，这个方法的功能是根据 viewContext 中的数据生成视图内容，并把生成的内容写到 writer 对象中。

全部完成以后我们还要注册自定义视图引擎，方法是在 Global.asax 文件中的 Application_Start()方法中使用 ViewEngines 类里的 Engines 集合来添加自定义视图引擎。当然，在添加之前我们可以先清理掉 ASP.NET MVC 默认的视图引擎。代码如下：

```
ViewEngines.Engines.Clear();
ViewEngines.Engines.Add(new MyViewEngine());
```



在上面的代码中，MyViewEngine 是自定义的一个视图引擎类。

好了，基本上要注意的问题都已经讲完了，下面来看一个实例。

8.4.2　实例描述

视图引擎比较简单，但是如果同时再举个实例应用，就会显得有点复杂。本实例使用一个

简单的博客文章页面，尽可能地将视图引擎与实例分离开讲，让大家更容易了解。

本实例基本上分为两部分：创建视图引擎以及实现博客页面并使用视图引擎。

8.4.3 实例应用

【例 8-4】博客文章页面。

(1) 打开前面我们使用的项目。

(2) 我们要先创建视图引擎，即先在 Models 目录中创建一个类文件 MyViewEngine.cs。

(3) 打开 MyViewEngine.cs 文件。因为要用到 StringTemplate 模板引擎，所以这里要先在页面中引用 StringTemplate 类的命名空间。代码如下：

```
using Antlr3.ST;
```

(4) 在 MyViewEngine.cs 文件中添加一个视图对象类 MyView，代码如下：

```
public class MyView : IView
{
    public string ViewPath { get; set; }
    public string MasterPath { get; set; }

    public MyView(string viewPath)
    {
        /*
         * 初始化视图文件路径
         */
        this.ViewPath = viewPath;
    }

    public MyView(string viewPath, string masterPath)
    {
        /*
         * 初始化视图文件路径和母版页路径
         */
        this.ViewPath = viewPath;
        this.MasterPath = masterPath;
    }

    public void Render(ViewContext viewContext, TextWriter writer)
    {
        HttpServerUtilityBase Server = viewContext.HttpContext.Server;

        //获得视图文件的物理路径
        string path = Server.MapPath(this.ViewPath);

        //读取文件的内容
        string content = File.ReadAllText(path);

        //创建模板引擎
        var template = new StringTemplate(content);

        //遍历 ViewData,设置模板引擎的值
        foreach (var data in viewContext.ViewData)
```

```
        {
            template.SetAttribute(data.Key, data.Value);
        }
        writer.Write(template.ToString());
    }
}
```

上面代码实现了在初始化 MyView 类对象的时候初始化视图文件的路径，然后在 Render()方法中获取指定的视图路径，读取视图文件并设置相应的值。

(5) 创建一个 MyViewEngine 类，实现视图引擎的选择。代码如下：

```
public class MyViewEngine : VirtualPathProviderViewEngine
{
    public MyViewEngine()
    {
        //初始化视图文件路径模板
        this.ViewLocationFormats = new string[]
        {
            "~/Views/{1}/{0}.page",
            "~/Views/Shared/{0}.page"
        };

        //初始化局部视图文件路径模板
        this.PartialViewLocationFormats = new string[]
        {
            "~/Views/{1}/{0}.modu",
            "~/Views/Shared/{0}.modu"
        };
        //初始化母版页文件路径模板
        this.MasterLocationFormats = new string[]
        {
            "~/Views/{1}/{0}.mast",
            "~/Views/Shared/{0}.mast"
        };
    }

    protected override IView CreatePartialView(ControllerContext
controllerContext, string partialPath)
    {
        return new MyView(partialPath);
    }

    protected override IView CreateView(ControllerContext controllerContext,
string viewPath, string masterPath)
    {
        return new MyView(viewPath, masterPath);
    }
}
```

上面代码在初始化视图引擎的时候同时初始化了视图、母版页和局部视图等类型视图文件的路径查询模板。



大家看到这里应该知道为什么默认视图引擎中的视图必须放在 Views 目录中了吧。如果要改变视图的存放目录，可以重定义一个视图引擎来实现。

(6) 就这两个类，视图引擎就声明完了。下面修改 Global.asax 文件中的 Application_Start() 方法，添加注册视图引擎的语句，代码如下：

```
protected void Application Start()
{
    AreaRegistration.RegisterAllAreas();
    RegisterRoutes(RouteTable.Routes);

    //注册视图引擎
    ViewEngines.Engines.Clear();
    ViewEngines.Engines.Add(new MyViewEngine());
}
```

至此，该视图引擎可以正常运行了。

下面我们简单写一个博客文章的例子测试一下。

(7) 在 Models 目录中添加一个类文件，命名为 BlogModels.cs。

(8) 在 BlogModels.cs 文件中添加 3 个实体类，分别用于封装页面、博客和文章信息，代码如下：

```
public class SitePage
{
    public string Copyright { get; set; }
    public string Title { get; set; }
}

public class Blog
{
    public string ChildTitle { get; set; }
    public string Title { get; set; }
}

public class Article
{
    public string Author { get; set; }
    public string Class { get; set; }
    public string Content { get; set; }
    public string Tag { get; set; }
    public DateTime Time { get; set; }
    public string Title { get; set; }
}
```

(9) 在 Controllers 目录中添加一个 Controller，命名为 BlogController。

(10) 修改 BlogController 的代码，如下所示：

```
public class BlogController : Controller
{
    public ActionResult Index()
    {
        ViewData["Page"] = this.GetPage();
        ViewData["Article"] = this.GetArticle();
        ViewData["Blog"] = this.GetBlog();

        return View();
    }

    private Article GetArticle()
    {
        return new Article()
```

```
        {
            Author = "张浩华",
            Class = "ASP.NET MVC",
            Content = "<p>    在 ASP.NET MVC 2 中，视图引擎的职责是根据参数选择并创建
视图对象，" +
            "视图引擎不负责视图内容的生成工作，视图内容的生成工作由视图对象来完成。" +
            "<p>    在 ASP.NET MVC 2 中，默认情况下，创建自定义的视图引擎需要实现
IViewEngine 接口" +
            "(该接口声明在 System.Web.Mvc 命名空间中)。" +
            "<p>    一般来说，几乎所有的模板引擎都会使用虚拟路径进行对视图文件的定位，并加
载视图。" +
            "所以如果我们使用 IViewEngine 的话，我们就还需要编写相应的路径处理代码。" +
            "<p>    VirtualPathProviderViewEngine 类是一个抽象类。该类包含了各种路径
的查询模板，" +
            "可以根据 controller 和 action 的值进行查询相应的视图文件。",
            Tag = "MVC,ASP.NET,自定义视图引擎",
            Time = new DateTime(2010, 11, 22),
            Title = "ASP.NET MVC 2 自定义视图引擎"
        };
    }

    private Blog GetBlog()
    {
        return new Blog()
        {
            Title = "张浩华",
            ChildTitle = "不要浮躁，冷静的思考，为人，为己……"
        };
    }

    private SitePage GetPage()
    {
        return new SitePage()
        {
            Copyright = "Copyright ©2010 张浩华 ",
            Title = "ASP.NET MVC 2 自定义视图引擎 - 张浩华 - 博客"
        };
    }
}
```

正常情况下，Blog、SitePage 以及 Article 类的实例的数据需要从数据库中获取，这里为了演示方便，使用 3 个方法创建 3 个对象直接返回。

在这里我们发现，以这种方式自定义的视图引擎不影响 Controller 中的任何操作，只需要改变视图和应用程序的视图引擎的配置即可。

(11) 在站点根目录下的 Views 目录中添加一个 Blog 子文件夹。

(12) 在/Views/Blog 目录中添加一个视图文件，命名为 Index.page。

(13) 修改 Index.page 文件，添加页面代码，如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>$Page.Title$</title>
    <link href="/Content/Blog.css" rel="stylesheet" type="text/css" />
</head>
<body>
<div id="main">
```

```
    <div id="header">
        <h1 class="title">$Blog.Title$</h1>
        <h2 class="child title">$Blog.ChildTitle$</h2>
    </div>
    <div id="body" >
        <div id="menu">博客园  社区  首页  新随笔  联系  管理  订阅</div>
        <div id="blog">
            <div class="title">$Article.Title$</div>
            <div class="content">
            $Article.Content$
            </div>
            <div class="tag">Tag 标签：$Article.Tag$</div>
            <div class="other">posted @
$Article.Time$ <span>$Article.Author$</span> 阅读(2) <span>评论(0)</span>
<span>编辑</span> <span>收藏</span> 所属分类：
<span>$Article.Class$</span></div>
        </div>
    </div>
    <div id="footer">
        $Page.Copyright$
    </div>
</div>
</body>
</html>
```

至此就完成了视图引擎的编写和测试代码的编写工作。下面来运行一下看看效果。

8.4.4 运行结果

运行项目，访问/Home/Index，结果如图 8-4 所示。



图 8-4　自定义视图引擎运行结果

在运行结果中，为了简化页面结构，页面右侧的信息被设置为页面背景。

8.4.5 实例分析



源码解析

本实例首先创建一个实现了 IView 接口的类 MyView，并且在 MyView 类中实现了 IView 接口中定义的方法 Render()，同时实现了视图的生成功能。创建一个自定义的视图引擎 MyViewEngine 类，它继承自 VirtualPathProviderViewEngine 类，实现了视图文件路径的定位和选择视图对象的功能。最后创建一个 BlogController 接受相应的请求，并在/Blog/Index 目录中添加一个视图文件 Index.page，用以保存视图信息。

8.5　使用母版页优化博客系统

上一节我们定义了一个视图引擎。是否觉得这才算视图引擎？

其实它还很简单。确实，开发一个视图引擎不是一件简单的事情。这里我们所写的视图引擎还很简陋。最起码，它不能实现代码复用，既没有导入局部视图，也不能使用母版页。

程序员一般都有一种追求完美的心态，本节我们来研究一下带母版页的视图引擎的使用方法。

**视频教学：**光盘/videos/08/8.5　使用母版页优化博客系统　　长度：4 分钟

8.5.1　实例描述

视图引擎的定义和使用需要大量代码，为了缩减篇幅，这里直接修改上一节中使用的项目，为其添加母版页的功能。

8.5.2　实例应用

【例 8-5】使用母版页优化博客系统。

(1) 打开前面我们使用的项目。

(2) 先来修改生成视图的类 MyView，代码如下：

```
public class MyView : IView
{
    public string ViewPath { get; set; }
    public string MasterPath { get; set; }

    public MyView(string viewPath)
    {
        /*
         * 初始化视图文件路径
         */
        this.ViewPath = viewPath;
    }

    public MyView(string viewPath, string masterPath)
    {
        /*
         * 初始化视图文件路径和母版页路径
```

```
         */
        this.ViewPath = viewPath;
        this.MasterPath = masterPath;
    }

    public void Render(ViewContext viewContext, TextWriter writer)
    {
        //获得模板引擎
        var viewTemplate = this.GetTemplate(this.ViewPath, viewContext);

        //如果设置了母版页，则将视图页内容加入母版页中，生成总的视图
        if (!string.IsNullOrEmpty(this.MasterPath))
        {
            var masterTemplate = this.GetTemplate(this.MasterPath,
viewContext);
            masterTemplate.SetAttribute("ChildView",
viewTemplate.ToString());
            viewTemplate = masterTemplate;
        }

        writer.Write(viewTemplate.ToString());
    }

    private StringTemplate GetTemplate(string viewPath, ViewContext
viewContext)
    {
        HttpServerUtilityBase Server = viewContext.HttpContext.Server;

        //获得视图文件的物理路径
        string path = Server.MapPath(viewPath);

        //读取文件的内容
        string content = File.ReadAllText(path);

        //创建模板引擎
        var template = new StringTemplate(content);

        //遍历 ViewData,设置模板引擎的值
        foreach (var data in viewContext.ViewData)
        {
            template.SetAttribute(data.Key, data.Value);
        }
        return template;
    }
}
```

这里主要修改了 Render()方法，该方法分别生成母版页和视图页的模板，然后使用视图页面的内容填充母版页的相应位置。

(3) 我们还需要将上一节所使用的视图文件分拆。在/Views/Shared 目录中创建一个视图文件，命名为 BlogMaster.mast，内容如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>$Page.Title$</title>
```

```
    <link href="/Content/Blog.css" rel="stylesheet" type="text/css" />
</head>
<body>
<div id="main">
    <div id="header">
        <h1 class="title">$Blog.Title$</h1>
        <h2 class="child title">$Blog.ChildTitle$</h2>
    </div>
    <div id="body" >
        <div id="menu">博客园  社区  首页  新随笔  联系  管理  订阅</div>
        <div id="blog">
        $ChildView$
        </div>
    </div>
    <div id="footer">
        $Page.Copyright$
    </div>
</div>
</body>
</html>
```

上面代码删除了页面主体区的内容，使用一个$ChildView$占位符，用于标识子页面的位置。

(4) 修改/Views/Blog/Index.page 文件，删除页面结构，只留主体区内容，代码如下：

```
<div class="title">$Article.Title$</div>
<div class="content">
$Article.Content$
</div>
<div class="tag">Tag 标签：$Article.Tag$</div>
<div class="other">posted @ $Article.Time$ <span>$Article.Author$</span> 阅读(2) <span>评论(0)</span> <span>编辑</span> <span>收藏</span> 所属分类：<span>$Article.Class$</span></div>
```

(5) 修改 BlogController 中的 Index 动作，让其在返回的时候指定视图和母版页。Index 动作代码如下：

```
public ActionResult Index()
{
    ViewData["Page"] = this.GetPage();
    ViewData["Article"] = this.GetArticle();
    ViewData["Blog"] = this.GetBlog();

    return View("Index", "BlogMaster");
}
```

至此，所有修改操作完成，我们可以直接运行项目查看结果。

这里的运行结果和上一节完全一样，所以不再截图显示了。

8.5.3　实例分析



源码解析

本实例首先修改视图生成对象 MyView 类，即修改 Render()方法，使其分别创建母版页和视图页的模板对象，然后使用视图页的模板对象将生成结果替换到母版页的相应位置，即可实

现视图的生成。之后还要将视图分开存放，即使母版页和内容页分离。最后在 Controller 中的 Action 中使用 View()方法的时候，指定母版页和视图页。

8.6 常见问题解答

8.6.1 自定义视图引擎和 WebForm 视图引擎能否共存



我自定义的视图引擎和 WebForm 视图引擎能不能共存？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

通过学习，我自定义了一个简单的视图引擎，确实没有 MVC 自带的强大，只能实现一些非常简单的功能。我想在系统中同时使用这两个视图引擎，那么我自定义的视图引擎和 WebForm 视图引擎能不能共存？

【解决办法】当然可以。

首先，你会发现注册视图引擎代码的操作方式很像一个集合，使用 Clear()方法清空，然后使用 Add 方法添加一个视图引擎即可。代码如下：

```
ViewEngines.Engines.Clear();
ViewEngines.Engines.Add(new MyViewEngine());
```

当把光标放在 ViewEngines 类的 Engines 对象上的时候会发现，Engines 是一个 ViewEngineCollection 类的对象。既然是一个集合，当然可以添加多个对象了。

当然，多个视图引擎要协同工作，执行顺序就会有先有后。这里采用和 URLRouting 一样的策略，即从头到尾遍历集合，找到匹配的第一个视图引擎，处理响应操作。在视图引擎选择上你可能需要做点调整。

现在第三方资料还不多，具体方法你可以去微软官方查询资料，MSDN 上“老赵”的视频好像提到过，你可以找找看。

8.6.2 什么时候需要自定义视图引擎



什么时候需要自定义视图引擎呢？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

在学习了 ASP.NET MVC 的自定义视图引擎以后，总觉得它很不实用，例如我要做一个简单的系统，很难实现复杂页面的输出，例如列表之类。

总感觉自定义的视图引擎要实现 WebForm 视图引擎那样的功能非常复杂。我什么时候会用到自定义视图引擎呢？

【解决办法】首先，学任何一门技术都不是一招就可以无敌的。

ASP.NET MVC 默认的 WebForm 视图引擎非常强大，如果我们实现了它的所有功能，当然会相当费时费力，而且基本上毫无价值可言，因为我们很难超越微软的技术团队。

我们要学它，是因为它是一门技术。掌握了它，看似无用，但也无所不用。第一，我们知道了 MVC 视图引擎的运和原理，对深入了解 MVC 框架和 ASP.NET 非常有益；第二，一些公司会根据自己的需要开发自己的视图引擎，掌握了它，你会更加容易地应用到工作中，节省了不少时间。

如果有一天你能做出一个非常强大的视图引擎，说不定微软的高薪聘书就到你家了。

8.7 习　　题

一、填空题

(1) 自定义的视图引擎要实现__________接口。

(2) 自定义一个可以使用母版页的视图引擎，然后在/View/User 目录中有一个 Create.page 视图文件和一个 UserMaster.mast 母版页文件，在 UserController 里的 Create 动作中我们要调用相应的视图页和母版页，请补充下面的代码。

```
public ActionResult Create()
{
    ViewData["User"] = this.GetUser();


}
```

(3) 使用 StringTemplate 模板引擎可以简化视图的生成，例如在 Controller 中有这样的一段代码：

```
var user = new { Name = "周星星" };
var template = new StringTemplate(content);
template.SetAttribute("User", user);
```

可以在视图文件中使用__________获取“周星星”这个名字。

二、选择题

(1) 这里定义了一个视图引擎，代码如下：

```
public class MyViewEngine :
{
    public MyViewEngine()
    {
        /* 初始化代码省略 */
    }

    protected override IView CreatePartialView(ControllerContext
controllerContext, string partialPath)
    {
        return new MyView(partialPath);
    }

    protected override IView CreateView(
```

```
        ControllerContext controllerContext, string viewPath, string
masterPath)
    {
        return new MyView(viewPath, masterPath);
    }
}
```

上面代码在横线处填入的内容是＿＿＿＿＿＿。

A. VirtualPathViewEngine　　　　B. VirtualPathProviderViewEngine

C. IviewEngine　　　　D. ProviderViewEngine

(2) 假如控制器 ProductController 下的 Index 动作对应站点根目录下的/Shitu/Product/Index.pg 视图文件，那么在使用视图引擎的构造方法初始化视图位置模板的时候需要这样配置：

```
this.ViewLocationFormats = new string[]
{

};
```

请补充上面代码。

A. "~/Shitu/{1}/{0}.pg"　　　　B. "~/Views/{1}/{0}.pg"

C. "~/Shitu/{0}/{1}.pg"　　　　D. "~/Views/{1}/{2}.pg"

(3) 在 ASP.NET MVC 中，自定义视图引擎需要使用的视图对象必须实现＿＿＿＿＿＿接口。

A. Views　　　　B. View

C. Ivews　　　　D. IView

(4) ASP.NET MVC 视图对象生成的结果是＿＿＿＿＿＿。

A. 一段 HTML 文本　　　　B. 一个 Web 文件

C. 一段 C#程序　　　　D. 一个 ASP.NET 页面

三、上机练习

上机练习： 在自定义视图中输出当前日期。

回顾一下自定义视图引擎，相信大家脑海中已经有一个十分清晰的认识了吧。本练习要求简单定义一个视图引擎，将路径/Time/Index 映射到 Views/Time/Index.page 视图文件中，实现在页面中打印当前日期，如图 8-5 所示。



图 8-5　显示当前日期



第 9 章　过　滤　器

内容摘要

前面章节已经介绍了控制器和控制器的动作，它们将负责调整用户、模型和视图之间的交互作用。给定的动作方法通常处理具体的用户与视图之间的交互作用。

例如，当用户单击页面上显示的产品的超链接，那么此次单击可能创建一个对名为 List 的 ProductsController 动作方法的请求，它将进入域对象，获取所有的产品，将其放在一起，并将结果显示给用户。

获取产品数据以及选择视图的过程是动作方法的主要职责，但它并不负责横向的关注点，例如日志记录、输出缓存、身份验证以及授权等。动作方法也不需要了解这些职责。按数学中的说法，它们与动作方法的用途互不相干。

这时需要使用到过滤器。过滤器是一种基于声明的特性，可以向动作方法提供横向行为。

学习目标

- 了解应用于 Action 和 Controller 的过滤器
- 了解如何规定页面的访问形式
- 了解缓存过滤器
- 了解异常过滤器
- 了解授权过滤器
- 了解自定义过滤器的方法

9.1 应用于 Action 的过滤器

应用于 Action 的过滤器，就是将已创建的过滤器应用到控制器中的动作方法。在项目开发过程中，开发人员习惯于使用这个过滤器来完善程序的严谨性。例如，当用户触发了某个事件的时候，可以及时处理异常，以保证项目的正常运行及使用。

**视频教学：**光盘/videos/09/9.1 应用于 Action 的过滤器 长度：7 分钟

9.1.1 基础知识——ActionFilter

Action 过滤器是通过继承 ActionFilterAttribute 类来实现的一个 Attribute 类。ActionFilterAttribute 是一个抽象类，提供了两个 virtual 方法 OnActionExecuting 和 OnActionExecuted；我们可以重写这两个方法。

ActionFilterAttribute 类的命名空间是 System.Web.Mvc，名字叫做动作筛选器，它继承于一个类和两个接口，这个类是 FilterAttribute，两个接口分别是 IActionFilter 和 IResultFilter。

可以使用动作筛选器特性来标记任何动作方法或控制器。如果要特别标记控制器，则动作筛选器将应用于该控制器中的所有动作方法。

ActionFilterAttribute 类包含几个需要重写的方法，如表 9-1 所示。

表 9-1 ActionFilterAttribute 类需要重写的方法

方 法	说 明
OnActionExecuted	在执行操作方法后由 MVC 框架调用
OnActionExecuting	在执行操作方法之前由 MVC 框架调用
OnResultExecuted	在执行操作结果后由 MVC 框架调用
OnResultExecuting	在执行操作结果之前由 MVC 框架调用

其中，参数 filterContext 表示筛选器上下文，动作过滤器的每个方法都具有具体的上下文对象，这些上下文对象向过滤器提供了信息，而在一些特殊情形下，允许过滤器取消动作。前两个方法具有与 IActionFilter 接口相同的签名，而后两个方法则来自 IResultFilter 接口。

在动作调用器已经找到动作方法之后，但是在调用动作方法之前，才会调用 OnActionExecuting。此时，控制器的上下文提供给了动作方法，TempData 字典则使用任何可能存在的临时数据来填充。

在调用了动作方法之后，但是在执行动作结果之前，调用器将调用 OnActionExecuted。因为所有的动作方法都返回一个动作结果，所以可以在分别通过 OnResultExecuting 和 OnResultExecuted 来执行结果前才调用控制器。

9.1.2 实例描述

我们日常生活中的衣、食、住、行，这些都要通过严谨的加工过程。比如衣服，它本来不

是衣服，是用布一步步缝出来的，如果衣服破了，我们可以把它修理一下，还能穿。

那么，本小节中将为大家讲解应用于 Action 的过滤器，这跟衣服的道理是一样的，我们可以不用把整件衣服都毁掉，直接将过滤器应用到控制器中的动作方法。

9.1.3 实例应用

【例 9-1】应用于 Action 的过滤器。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名字为 Action_filter。

(2) 在 Home 控制器里面创建一个过滤器，名字叫 TestFilter。该过滤器继承自 ActionFilterAttribute 类，该类定义了 4 个方法，分别在执行动作方法和执行动作结果前后，由 MVC 框架调用。实现代码如下：

```
 public class TestFilter : ActionFilterAttribute
 {
     public override void OnActionExecuting(ActionExecutingContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnActionExecuting<br/>";
     }
     public override void OnActionExecuted(ActionExecutedContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnActionExecuted<br/>";
     }
     public override void OnResultExecuting(ResultExecutingContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnResultExecuting<br/>";
     }
     public override void OnResultExecuted(ResultExecutedContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnResultExecuted<br/>";
     }
 }
```

(3) 将此过滤器应用于 Home 控制器的 Index()方法，实现代码如下：

```
 [TestFilter]
 public ActionResult Index()
 {
     ViewData["Message"] = "欢迎使用 ASP.NET MVC!";
     return View();
 }
```

(4) 在视图中读取 Session 的值，实现代码如下：

```
<%=Session["temp"] += "View Execute<br/>"%>
```

9.1.4 运行结果

运行程序，效果如图 9-1 所示。

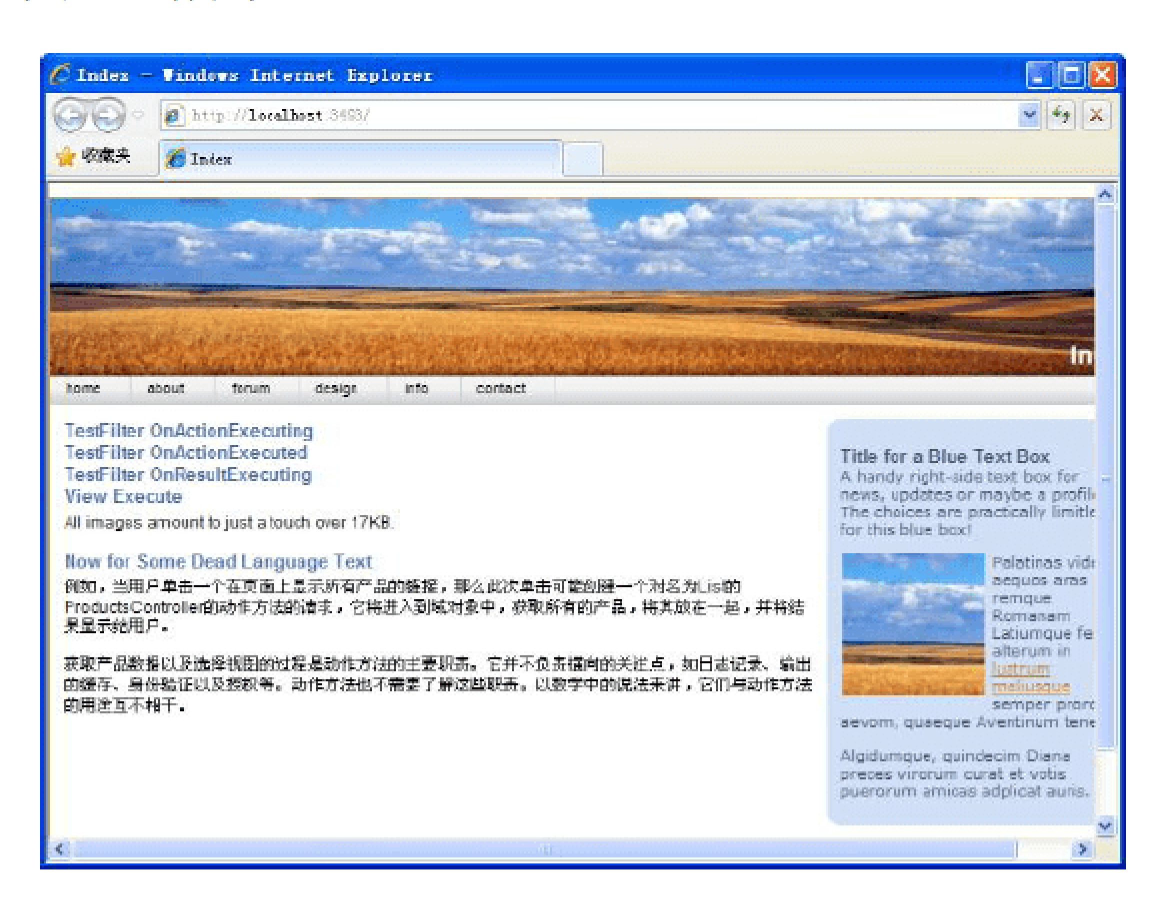


图 9-1 应用于 Action 的过滤器

9.1.5 实例分析



源码解析

这里最关键的就是创建继承于 ActionFilterAttribute 的 TestFilter 类，并重写 ActionFilterAttribute 类中的方法，然后把 TestFilter 类应用到 Home 控制器的 Index()方法上。实现代码如下：

```
[TestFilter]
public ActionResult Index()
```

最后只要在视图中读取 Session 的值即可。

9.2 应用于 Controller 的过滤器

应用于 Controller 的过滤器就是将已创建的过滤器应用到控制器上，这将运用该控制器中所有的动作方法。同前面讲到的应用于 Action 的过滤器类似，只不过应用的对象不一样罢了。

**视频教学：**光盘/videos/09/9.2 应用于 Controller 的过滤器 长度：6 分钟

9.2.1 基础知识——过滤 Controller 的方法

将 Filter 应用到 Controller 上包括以下两种方式。

(1) 直接将过滤器应用在 Controller 上，例如：

```
[TestFilter]
public class HomeController:Controller{ }
```

(2)重写 Controller 内的 OnActionExecuting、OnActionExecuted、OnResultExecuting 和 OnResultExecuted 四个方法。

在该案例中，我们将以第一种方式为例，讲解如何将过滤器应用于 Controller。

9.2.2　实例描述

前面我们讲了应用于 Action 的过滤器，与本节我们要讲的应用于 Controller 的过滤器相比，其实道理是一样的，只不过应用的范围有大有小罢了。

下面我们一起来看实现的效果就明白了。

9.2.3　实例应用

【例 9-2】应用于 Controller 的过滤器。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名字叫 Control_filter。

(2) 在 Home 控制器里面创建一个过滤器，名字叫 TestFilter。该过滤器继承自 ActionFilterAttribute 类，该类定义了 4 个方法，分别在执行动作方法和执行动作结果前后，由 MVC 框架调用。实现代码如下：

```
 public class TestFilter : ActionFilterAttribute
 {
     public override void OnActionExecuting(ActionExecutingContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnActionExecuting<br/>";
     }
     public override void OnActionExecuted(ActionExecutedContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnActionExecuted<br/>";
     }
     public override void OnResultExecuting(ResultExecutingContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnResultExecuting<br/>";
     }
     public override void OnResultExecuted(ResultExecutedContext
filterContext)
     {
         filterContext.HttpContext.Session["temp"] += "TestFilter
OnResultExecuted<br/>";
     }
 }
```

(3) 将此过滤器应用于 Home 控制器，并在 Index()和 About()方法中设置 ViewData 的值，实现代码如下：

```
    [TestFilter]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Message"] = "欢迎使用 ASP.NET MVC!";
            return View();
        }
        public ActionResult About()
        {
            ViewData["Message"] = "欢迎来到 About 视图";
            return View();
        }
    }
```

(4) 分别在 Home 控制器的两个视图中读取 ViewData 和 Session 的值，实现代码如下：

```
<%:ViewData["Message"] %><br />
<%=Session["temp"] += "View Execute<br/>"%>
```

9.2.4 运行结果

运行程序，页面默认显示 Home 控制器对应的 Index 视图，效果如图 9-2 所示。

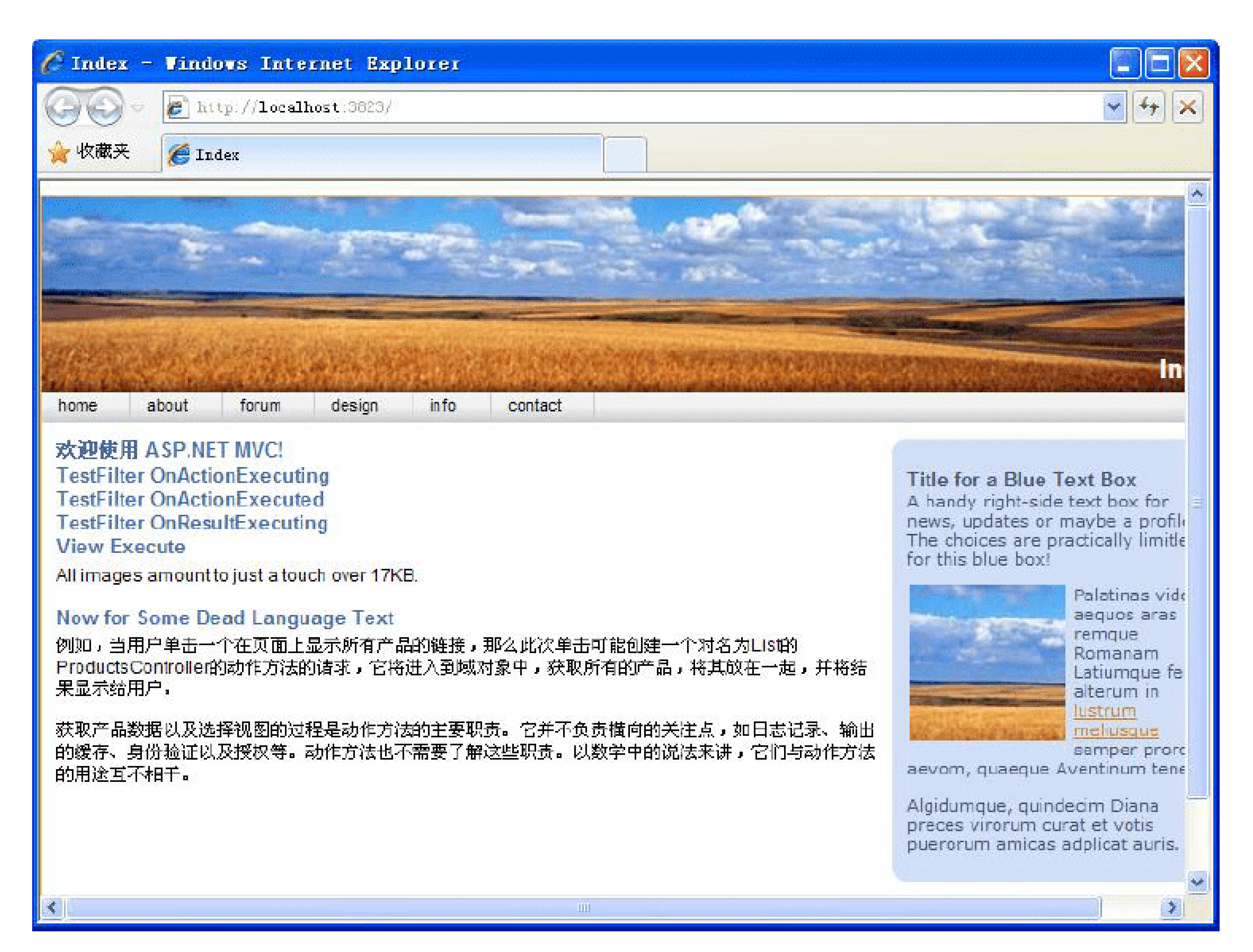


图 9-2　Index 视图

当在地址栏中输入 http://localhost:3823/home/about 并回车后，效果如图 9-3 所示。

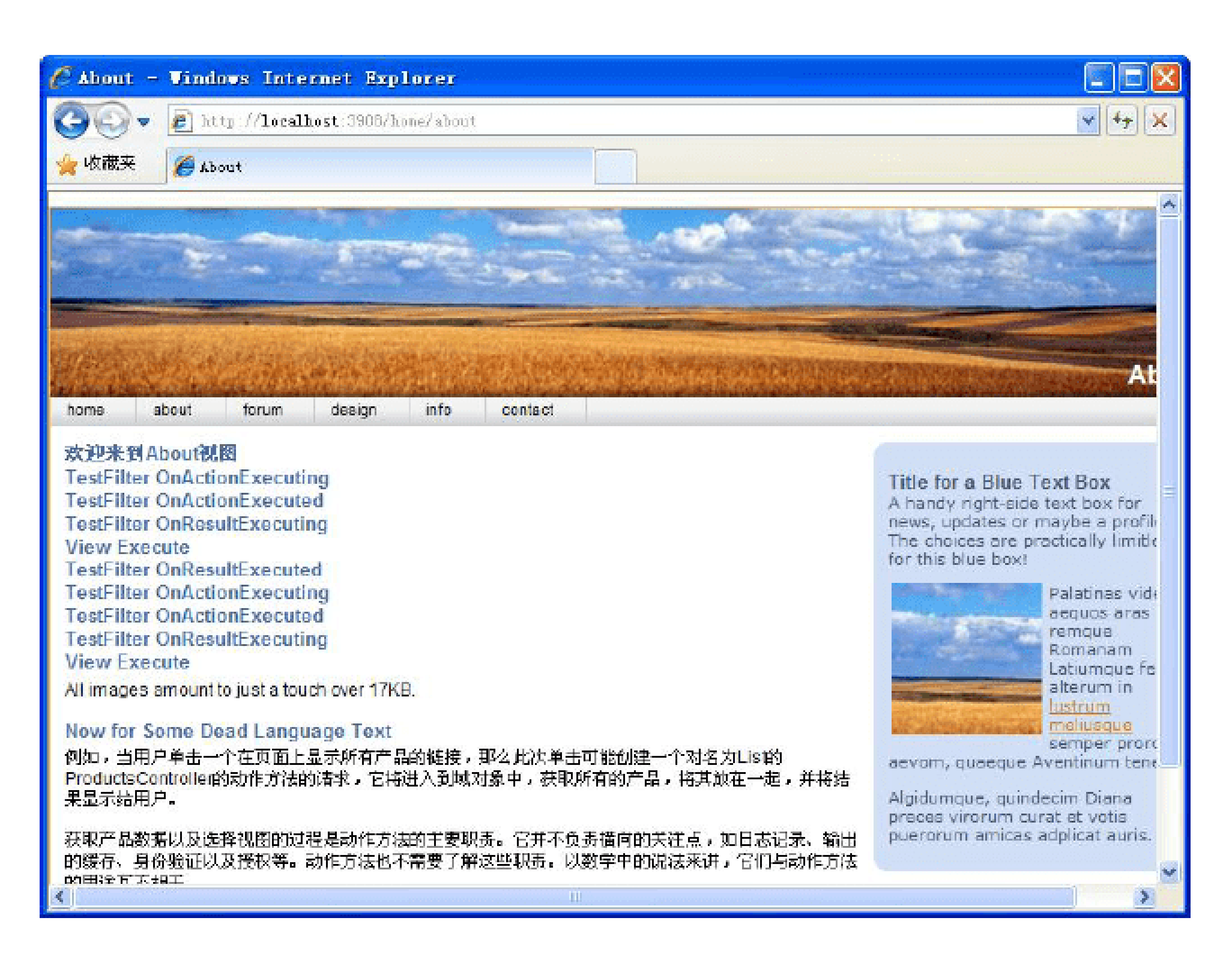
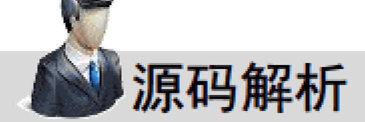


图 9-3　About 视图

9.2.5　实例分析



源码解析

这里最关键的就是创建继承自 ActionFilterAttribute 的 TestFilter 类，并重写 ActionFilterAttribute 类中的方法，然后把 TestFilter 类应用到 Home 控制器上。实现代码如下：

```
[TestFilter]
public class HomeController : Controller
```

最后只需要在视图中读取 Session 的值即可。

9.3　规定页面的访问形式

规定页面的访问形式就是指定某个页面只能以某种访问形式被访问。在 Web 项目开发中，如果要设置页面的访问形式，可以直接设置 Form 的 method 属性值为 POST 或 GET。然而，ASP.NET MVC 框架中却有不同的设置规则。

本节主要讲解 AcceptVerbs 类和 HttpVerbs 枚举的用法，可以用来规定页面的访问形式。

**视频教学**：光盘/videos/09/9.3　规定页面的访问形式　　长度：6 分钟

9.3.1　基础知识——AcceptVerbs 类和 HttpVerbs 枚举

在访问数据的时候，一般有两种方式：GET 和 POST。过滤器也可以指定页面的访问形式，需要用 AcceptVerbs 类来实现。

AcceptVerbs 类表示一个特性，该特性的操作方法将响应 HTTP 谓词。规定页面的访问形式代码如下：

```
[AcceptVerbs(HttpVerbs.Post)]
```

其中，HttpVerbs 表示 HTTP 谓词，枚举值如表 9-2 所示。

表 9-2　HttpVerbs 枚举值

枚举值	说　明
Get	检索由请求 URI 标识的信息或实体
Post	发布新实体作为对 URI 的补充
Put	替换由 URI 标识的实体
Delete	请求删除指定的 URI
Head	检索由请求 URI 标识的信息或实体的消息头

9.3.2　实例描述

我们访问一个网站，通常是先访问这个网站的首页。在首页中单击一个链接的时候，我们就可以访问到另一个页面，这叫做 POST 访问形式，也是最常用的访问形式。然而，我们也可以直接输入网址来访问指定页面。

下面这个案例将讲解如何规定页面的访问形式。

9.3.3　实例应用

【例 9-3】规定页面的访问形式。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Accept_Page。

(2) 在 Home 控制器的 Index 视图里写一个简单的用户登录页面，并设置用户名文本框 name 属性值为 username，密码框 name 属性值为 pwd，最后创建一个【登录】按钮。当用户单击【登录】按钮之后，页面跳转到另外一个视图。实现代码如下：

```
<% using (Html.BeginForm("Check","Home"))
   { %>
 用户名：<input id="username" name="username" type="text" /><br />
 密码：<input id="pwd" name="pwd" type="password" /><br />
 <input id="Btn login" type="submit" value="登录" />
 <%} %>
```

(3) 在 Home 控制器中定义一个名叫 Check 的动作方法，用于验证用户登录是否成功，并且设置提交方式只能是 POST 方式。实现代码如下：

```
 [AcceptVerbs(HttpVerbs.Post)]                     //提交方式只能是 POST
 public ActionResult Check()
 {
     string username=Request.Form["username"]; //获取表单中用户名
     string pwd=Request.Form["pwd"];           //获取表单中密码
```

```
    if (username == "admin" && pwd == "admin")     //判断用户名和密码是否匹配
    {
       ViewData["Message"] = "用户名: "+username+",密码: "+pwd;
       return View();
    }
    else
    {
       ViewData["Message"] = "check faile";
    }
    this.ModelState.AddModelError("Message", "sdf");
    return View();
}
```

(4) 添加 check 视图，获取用户名和密码的值。实现代码如下：

```
<%=ViewData["Message"]%>
```

(5) 在 Global.asax 文件中设置路由，要写到默认路由的前面。实现代码如下：

```
routes.MapRoute(
    "Default2",     // 路由名称
    "{controller}  /{action}/{username}/{pwd}", // 带有参数的 URL
    new {  }        // 参数默认值
);
```

9.3.4　运行结果

运行程序，效果如图 9-4 所示。

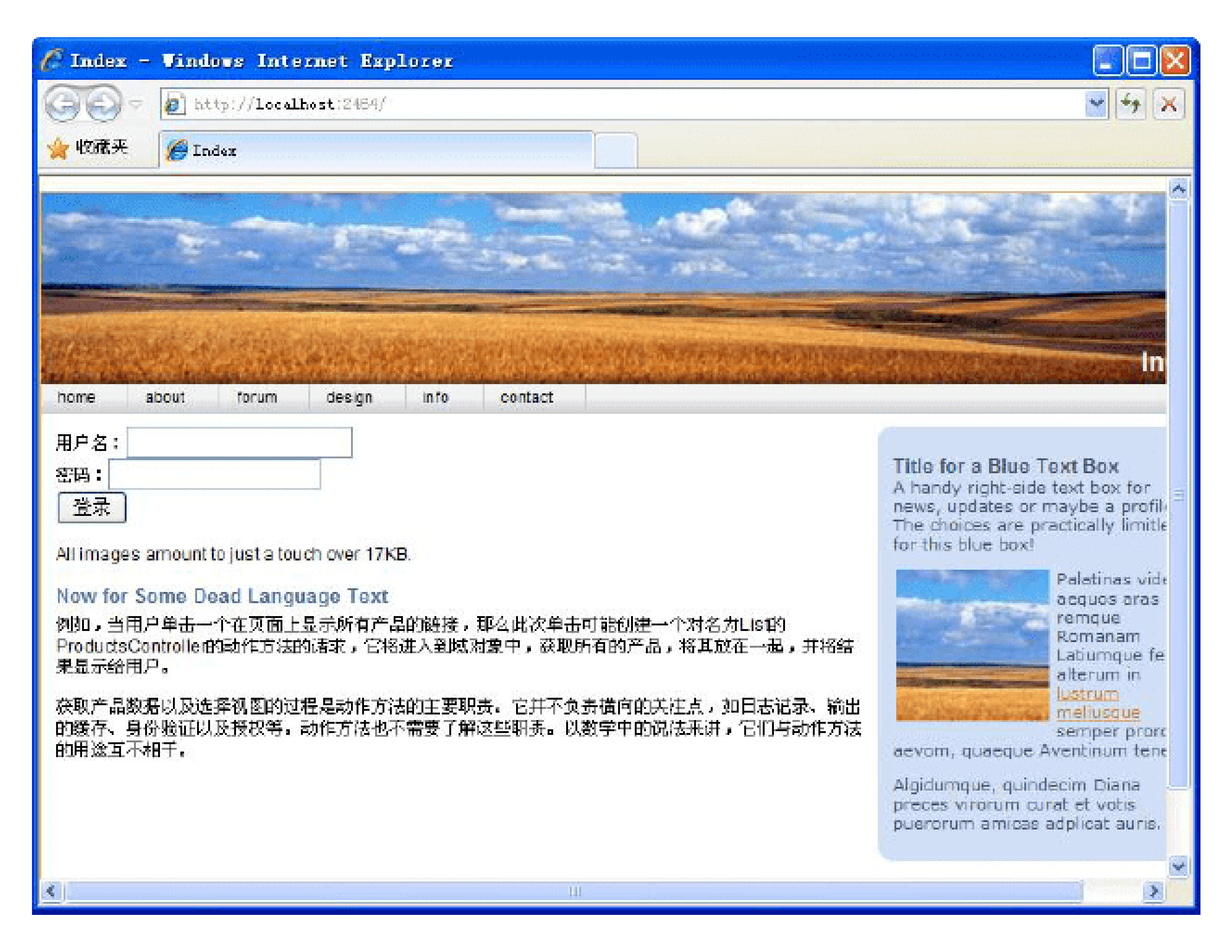


图 9-4　用户登录界面

输入用户名 admin，密码 admin，然后单击【登录】按钮，将以 POST 方式提交到 check 页面，并显示出用户名和密码，效果如图 9-5 所示。

如果用户没有在文本框里输入用户名和密码，而是在地址栏里直接访问 Check 页面，也就是以 GET 方式访问页面，那么页面将会报错，效果如图 9-6 所示。



图 9-5　登录之后的页面



图 9-6　应用程序报错

9.3.5　实例分析



源码解析

该案例主要讲解如何规定页面的访问形式。这里只限制被访问页面以 POST 方式访问，实现代码如下：

```
[AcceptVerbs(HttpVerbs.Post)]
```

用 Response.Form 方法获取被提交表单中的文本框的值。

9.4　规定 Action 的名称

规定 Action 的名称就是为指定的动作方法设置 Action 名。可以使用 ActionName 对象为控制器动作方法重新指定一个名称，在访问这个动作方法的时候将重新定位到这个指定的名称，即系统只识别这个指定的动作名称。

 **视频教学：**光盘/videos/09/9.4　规定 Action 的名称　　长度：4 分钟

9.4.1　基础知识——ActionName

ActionNameAttribute 表示一个用于操作名称的特性。如果不想用方法名作为 Action 名，或 Action 名为关键字，那么可以使用 ActionName 为动作方法指定一个 Action 名。例如以下代码：

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Example()
{
return View();
}
```

9.4.2　实例描述

我们聊天所用的 QQ，它可以有多个属性，比如说 QQ 号码、QQ 昵称等，但是不管有几个属性，它们都指向一个身份，就是你的 QQ，也就是你。还有作家，他们有自己的名字，也有自己的笔名。这些多对一的情况，还是很多的。

本节将讲解如何为指定的动作方法名指定一个 Action 名称。

9.4.3　实例应用

【例 9-4】规定 Action 的名称。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 ActionName_Page。

(2) 在 Home 控制器下面创建一个动作方法 User，但是不想用这个方法名作为 Action 名，那么我们可以给它重新指定一个。例如以下代码：

```
[ActionName("username")]
public ActionResult User()
{
    ViewData["UserName"] = "this is username!";
    return View();
}
```

(3) 为这个 Action 名为 username 的动作方法创建一个对应的视图，名字也是 username，并且在该视图中获取 ViewData 的值。实现代码如下：

```
<%=ViewData["UserName"] %>
```

9.4.4　运行结果

运行程序，在地址栏中输入 http://localhost:3758/home/username 并回车，效果如图 9-7 所示。

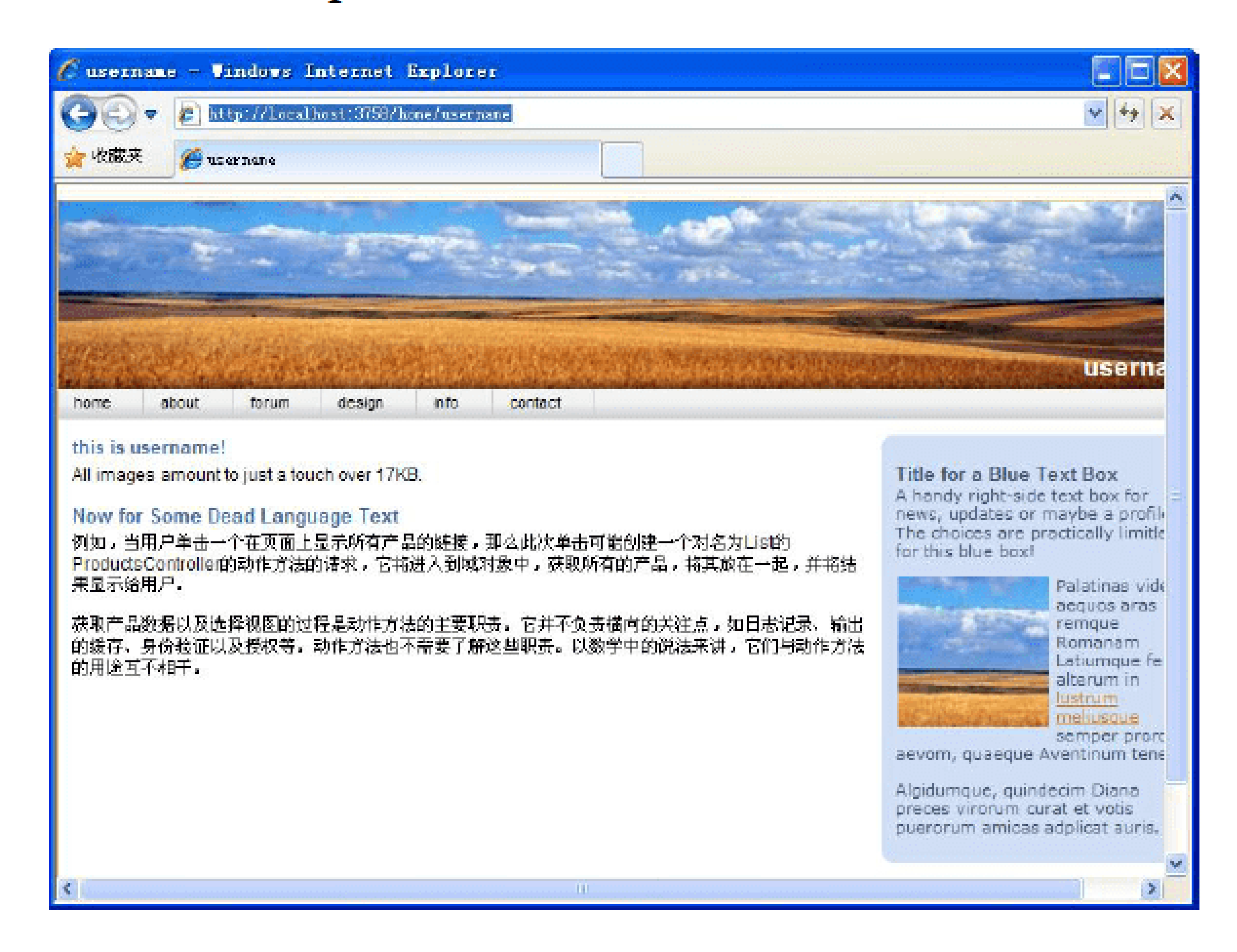


图 9-7　规定 Action 的名称

9.4.5　实例分析



源码解析

该案例为 User 动作方法设置了 Action 名，创建的视图要和这个 Action 名同名。当你要访问这个动作方法的时候，就通过这个 Action 名称去访问，实现代码如下：

```
[ActionName("username")]
public ActionResult User()
```

在地址栏中输入的地址为~home/username。

9.5　缓存当前时间

OutputCacheAttribute 用来缓存动作方法的输出。该特性是到 ASP.NET 输出缓存功能内的连接，而且提供了在使用@OutputCache 页面指令时得到大部分相同的 API 和行为。

因为输出缓存是 ASP.NET 非常著名的功能，所以本节没有深入讨论缓存行为自身的细节，而是将注意力集中于 MVC 的实现上。此时，可能会有一个问题，为什么不在视图中只使用@OutputCache 指令呢？

对于 MVC 而言，在选中视图之前，首先执行控制器动作。将输出缓存放到视图上，实际上就是将缓存视图输出，而动作方法仍然在每个请求上执行，这样就否定了缓存输出的大部分优点。

通过将该特性运用到动作方法中，过滤器随后可以确定缓存是否有效并跳过动作方法的调

用，直接呈现缓存的内容。

**视频教学：**光盘/videos/09/9.5　缓存当前时间　　长度：7 分钟

9.5.1　基础知识——OutputCache

表 9-3 列出了 OutputCacheAttribute 类的属性。这些设置用来控制 OutputCache 过滤器如何执行其缓存动作。

表 9-3　OutputCacheAttribute 类的属性

属　性	说　明
CacheProfile	即将使用的缓存设置的名称，允许将缓存的配置放到 web.config 文件中而不是特性中。随后，该特性可以通过该属性来引用配置设置
Duration	指定了将输出存储到缓存中的秒数
Location	指定了可能缓存内容的地方。枚举 OutputCacheLocation 包含允许的位置：Any、Client、Downstream、Server、None 和 ServerAndClient 等
NoStore	将 HTTP 题头 Cache-Control: Private, no-store 设置为阻止浏览器缓存响应，等价于调用 Response.Cache.SetNoStore
SqlDependency	特殊格式化的字符串值，包含一组输出缓存所依赖的数据库和表的名称对。当这些表中的数据发生了更改时，则缓存失效
VaryByContentEncoding	在.NET Framework 3.5 中引入，这是以逗号分隔的内容编码的列表，用于变更缓存
VaryByCustom	确定是否基于对 Global.asax.cs 文件中 GetVaryByCustomString 的调用缓存新版本的输出，为开发人员提供了对缓存的完全控制
VaryByHeader	基于 http 题头改变缓存。例如，可能使用它基于 Accept-Language 题头缓存不同版本的输出

9.5.2　实例描述

当你访问一个网站的时候，服务器将会把你的信息缓存起来。如果你再次访问该网站，很快就能将页面显示出来。这样看来，银行系统是最严谨的，如果你 5 分钟之内没有进行任何操作，就会自动退出系统。也就是将缓存数据删除。

下面这个例子将讲解如何缓存当前时间。

9.5.3　实例应用

【例 9-5】缓存当前时间。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Cache_page。

(2) 在 Home 控制器里创建一个动作方法 Cache_action，并设置该动作方法的缓存时间

Duration 的值。实现代码如下：

```
[OutputCache(Duration=1,VaryByParam="none")]
public ActionResult Cache action()
{
    ViewData["Data"] = "sjfjdfjshjfhjdh";
    return View();
}
```

9.5.4　运行结果

运行程序，在地址栏中输入 http://localhost:4588/home/cache_action 并回车，效果如图 9-8 所示。



图 9-8　缓存当前时间

当你一直不停地刷新页面的时候，页面数据将在 10 秒之后改变，也就是缓存当前时间 10 秒之后，页面才会刷新，效果如图 9-9 所示。



图 9-9　缓存之后的页面刷新

9.5.5 实例分析



源码解析

这个案例很简单，只需要设置缓存的时间即可。OutputCache 是缓存类，设置缓存的时间可以使用 Duration 属性(例如 20，表示缓存 20 秒)。设置完缓存之后，可以不断地刷新页面，20 秒之内，页面不会有任何变动。20 秒过后，页面就会重新获取当前时间。

9.6 异常过滤器

异常过滤器是一种比较常见的过滤器。当页面出错时，用于处理错误信息。在 Web 项目开发过程中，通常会将异常过滤器应用于动作方法上。



视频教学：光盘/videos/09/9.6 异常过滤器　　长度：6 分钟

9.6.1 基础知识——HandleError

HandleErrorAttribute 是包含在 ASP.NET MVC 中的默认异常过滤器。如果动作方法抛出一个未处理的异常，且该异常与指定的异常类型匹配或源自它，那么就可以使用该过滤器来指定需要处理的异常类型以及需要显示的视图(如果需要，还包括主视图)。

默认情况下，如果没有指定异常类型，那么过滤器将处理所有的异常。如果没有指定视图，那么过滤器将默认处理名为 Error 的视图。默认的 ASP.NET MVC 项目在 Shared 文件夹中包含了一个名为 Error.aspx 的视图。

请看下面的示例：

```
[HandleError(ExceptionType=typeof(ArgumentException),View="ArgError")]
public ActionResult GetProduct(string name)
{
 if(name == null)
  {
    throw new ArgumentNullException("name");
  }
  return View();
}
```

因为 ArgumentNullException 继承自 ArgumentException 类，所以将 null 传递到该动作方法中将导致显示 ArgError 视图。

在某些情况下，可能希望将多个异常过滤器运用到同一个动作方法中。为了将最具体的异常类型放在最前面而将较不明确的放在后面，此时需要指定这些情况的顺序，这是很重要的。例如，在下面的代码片段中：

```
//This is WRONG!
[HandleError(Order=1, ExceptionType=typeof(Exception))
[HandleError(Order=2,
ExceptionType=typeof(ArgumentException),View="ArgError")]
public ActionResult GetProduct(string name)
{
  …
}
```

第一个过滤器要比其后的过滤器更为通用，它将处理所有的异常，而始终不会给第二个过滤器任何机会来处理异常。为了修复此问题，只需将过滤器从最具体到最不具体进行排序即可。

```
//This is BETTER!
[HandleError(Order=1,
ExceptionType=typeof(ArgumentException),View="ArgError")]
[HandleError(Order=2, ExceptionType=typeof(Exception)]
public ActionResult GetProduct(string name)
{
  …
}
```

当该异常过滤器处理一个异常时，它创建了一个 HandleErrorInfo 类的实例并设置在呈现 Error 视图时 ViewDataDictionary 实例的 Model 属性。表 9-4 展示了 HandleErrorInfo 类的属性。

表 9-4 HandleErrorInfo 类的属性

属 性	说 明
Action	抛出异常的动作的名称
Controller	在其中抛出异常的控制器的名称
Exception	抛出的异常



该过滤器没有捕捉到在没有启用自定义错误时位于调试构建中的异常。过滤器只是检查 HttpContext.IsCustomErrorEnabled 以确定是否处理该异常。这样做的理由是，允许通过信息更丰富的 Yellow Screen of Death 来显示开发阶段与异常有关的信息。

9.6.2 实例描述

为了确保项目能够正常运行，通常我们会用 try 语句捕获异常，这是一种处理异常的方式。在本节中，我们讲的是异常过滤器就是用过滤器捕获异常。

9.6.3 实例应用

【例 9-6】异常过滤器。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Error_filter。

(2) 在 Home 控制器中创建一个处理异常的动作方法，实现代码如下：

```
[HandleError(ExceptionType = typeof(ArgumentException), View = "ArgError")]
public ActionResult GetProduct(string name)
{
    if (name == null)
    {
        throw new ArgumentNullException("name");
    }
    return View();
}
```

(3) 在 Global.asax 文件中设置对应的路由，并写在默认路由之前。实现代码如下：

```
routes.MapRoute(
    "Default2", // 路由名称
    "{controller}/{action}/{name}", // 带有参数的 URL
    new { } // 参数默认值
);
```

9.6.4　运行结果

运行程序，在访问 Home 控制器的 GetProduct 动作方法时，如果没有传递 name 参数，那么页面将输出异常，即 URL 地址为 http://localhost:2562/home/getproduct，效果如图 9-10 所示。

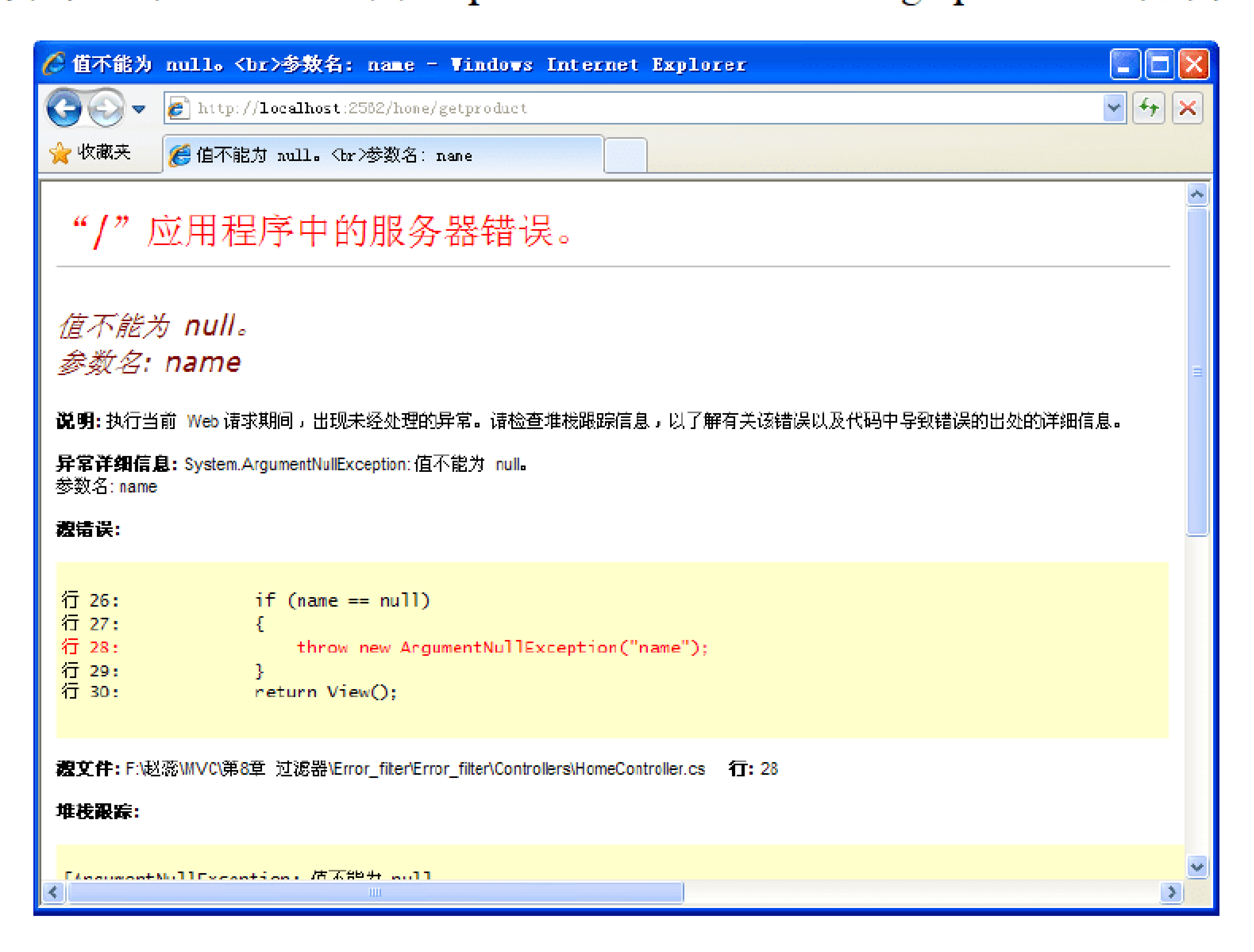


图 9-10　未异常处理的页面

反之，如果传入 name 参数，将没有异常，效果如图 9-11 所示。

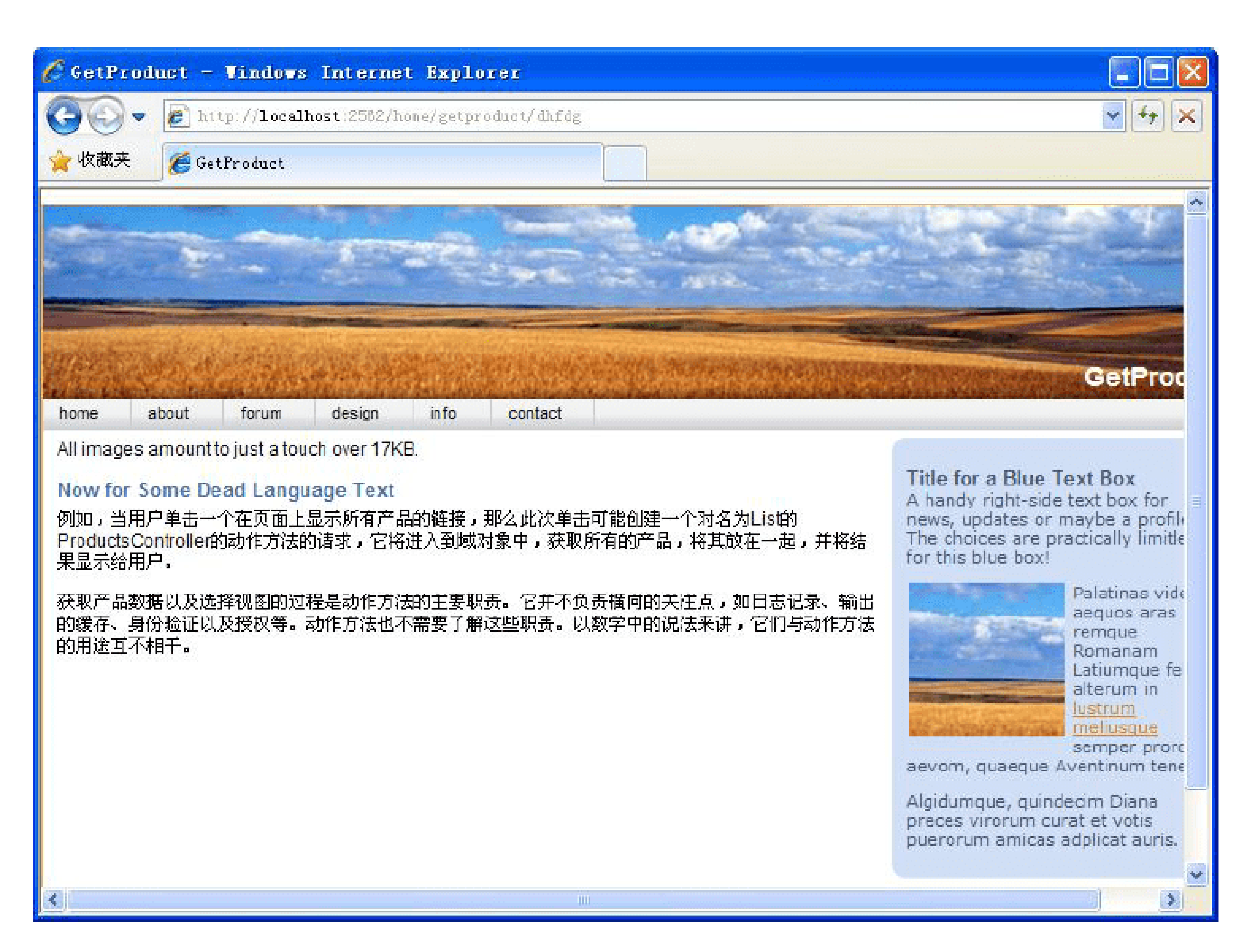


图 9-11　没有异常的页面

9.6.5　实例分析

**源码解析**

使用 HandleError 捕获异常。可以设置多个属性(例如 View 属性)，当页面有异常的时候，就跳转到异常页面。

在这个案例中，创建了一个动作方法，当访问这个动作出错的时候，就会捕获异常。实现代码如下：

```
[HandleError(ExceptionType = typeof(ArgumentException), View =
"ArgError")]
```

9.7　授权过滤器

授权过滤器用于限制对控制器或控制器动作的访问。在每个项目的开发过程中，一定要考虑权限问题，通常会将权限写入数据库中，然后判断用户权限，这是比较普通的方法。然而，在 ASP.NET MVC 中，可以使用授权过滤器来操作用户的访问权限。

**视频教学：**光盘/videos/09/9.7　授权过滤器　　　　长度：7 分钟

9.7.1　基础知识——Authorize

AuthorizeAttribute 是包含在 ASP.NET MVC 中默认的授权过滤器。可以使用它来限制对动

作方法的访问。将该特性运用到控制器上，可以迅速地将其运用到每个动作方法中。

在运用该过滤器时需要牢记如下内容。

- 在运用该特性时，可以指定一个逗号来划分角色(Role)或用户(User)的列表。如果指定了一个角色的列表，那么为了执行动作方法，用户必须是列表中的成员。同样，如果指定了一个用户列表，那么当前用户的名称必须在该列表中。
- 如果没有指定任何角色或用户，那么为了调用动作方法，必须验证当前用户。这是阻止非验证用户访问特殊控制器动作的一个简单方法。
- 如果用户试图访问运用了该特性的动作方法且在授权检查中失败，那么过滤器将引发服务器返回一个 401 Unauthorized 的 HTTP 状态代码。
- 对于启用了表单验证且在 web.config 中指定了注册 URL 的情形，ASP.NET 将处理该响应代码并将用户重新引向注册页面。这是 ASP.NET 已有的行为，对于 ASP.NET MVC 也不是新鲜事物。

9.7.2　实例描述

一个企业里面有许许多多的员工，由于每个人的工作性质或者职位不同，那么他们的工作权限也不同。这就是我们本节要讲到的授权问题。

9.7.3　实例应用

【例 9-7】授权过滤器。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Author_filter。

(2) 在 Home 控制器里创建一个动作方法 Admin_action，并设置该动作方法只有 Admins 角色能够访问。实现代码如下：

```
[Authorize(Roles="Admins")]
public ActionResult Admin_action()
{
    ViewData["Date"] = DateTime.Now;        //获取当前时间
    return View();
}
```

(3) 在 Admin_action 视图中读取 ViewData 的值。实现代码如下：

```
<%=ViewData["Date"] %>
```

9.7.4　运行结果

运行程序，当访问 Home 控制器下的 Admin_action 动作方法时，页面会自动跳转到登录页面。输入用户名和密码，如果该用户名所属的角色为 Admins，那么该用户就可以访问该动作方法，效果如图 9-12 所示。



图 9-12　授权过滤器

9.7.5　实例分析



源码解析

该案例是要访问 Home 控制器里的 Admin_action 动作方法，但不是所有的用户都能够访问。这里为该动作方法授权，即只有所属角色为 Admins 的用户才能够访问该动作方法。实现代码如下：

```
[Authorize(Roles="Admins")]
```

当然，我们也可以将它应用到控制器上，这样该控制器就被授权了。

9.8　自定义动作过滤器

前面介绍了包含在 ASP.NET MVC 中的过滤器。每个过滤器都是不同过滤器类型的实例。AuthorizeAttribute 是授权过滤器的一个实例，OutputCacheAttribute 是动作/结果过滤器的一个实例，而 HandleError 是异常过滤器的一个实例。在本节中，将为大家讲解如何自定义一个过滤器。



视频教学：光盘/videos/09/9.8　自定义动作过滤器　　长度：5 分钟

9.8.1　基础知识——自定义过滤器

每个过滤器类型都通过一个接口来连接。具体而言，实现过滤器的特性必须编写继承自 FilterAttribute 的类并实现如下 4 个接口中的之一。

- IauthorizationFilter
- IactionFilter
- IresultFilter
- IexceptionFilter

动作过滤器和结果过滤器是最常见的过滤器类型。如果发现在执行动作方法的前后或者在执行动作方法结果的前后需要了解一些自定义的逻辑，那么这些就是可能实现的两种类型的过滤器。

事实上，它们都非常相似，以致 MVC 小组包含一个继承了 FilterAttribute 且实现了两个接口的基类 ActionFilterAttribute。在编写动作过滤器或结果过滤器时(两种过滤器的组合)，只继承 ActionFilterAttribute 将更为容易。

9.8.2　实例描述

以前我们做项目的时候，有系统自带的用户控件，当然我们也可以自定义用户控件，这样使用起来更加方便。同样，过滤器也可以自定义。

接下来将把关于动作过滤器的知识运用到实际中，并编写一个动作过滤器。

9.8.3　实例应用

【例 9-8】自定义动作过滤器。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Users_filter。

(2) 添加一个名为 TimerAttribute 的类，该类继承自 ActionFilterAttribute 类。实现代码如下：

```
using System.Diagnostics;
using System.Web.Mvc;
  public class TimerAttribute:ActionFilterAttribute
  {
      public TimerAttribute()
      {
          this.Order = int.MaxValue;
      }
      public override void OnActionExecuting(ActionExecutingContext
filterContext)
      {
          var controller = filterContext.Controller;
          if (controller != null)
          {
              var stopwatch = new Stopwatch();
              controller.ViewData[" StopWatch"] = stopwatch;
              stopwatch.Start();
          }
      }
      public override void OnActionExecuted(ActionExecutedContext
filterContext)
      {
          var controller = filterContext.Controller;
          if (controller != null)
          {
```

```
            var stopwatch=(Stopwatch)controller.ViewData[" StopWatch"];
            stopwatch.Stop();
            controller.ViewData[" Duration"] =
stopwatch.Elapsed.TotalMilliseconds;
          }
      }
  }
```

(3) 下面来测试 Timer 过滤器。在调用动作方法之前，该过滤器简单地添加和启动了 ViewData 中的 Stopwatch 类的一个实例。在调用动作方法之后，过滤器检索到秒表(Stopwatch)的实例，并将已用时间添加到 ViewData 字典中。

此时，可以将这个新的动作过滤器应用到动作方法中。在该案例中，为了让这个展示更有趣，动作方法将暂停一个随机的毫秒时间。

```
[Timer]
public ActionResult Index()
{
    ViewData["Title"] = "Home Page";
    ViewData["Message"] = "欢迎使用 ASP.NET MVC!";
    var rnd = new Random();
    int randomNumber = rnd.Next(30);
    Thread.Sleep(randomNumber);
    return View();
}
```

9.8.4 运行结果

运行程序，效果如图 9-13 所示。

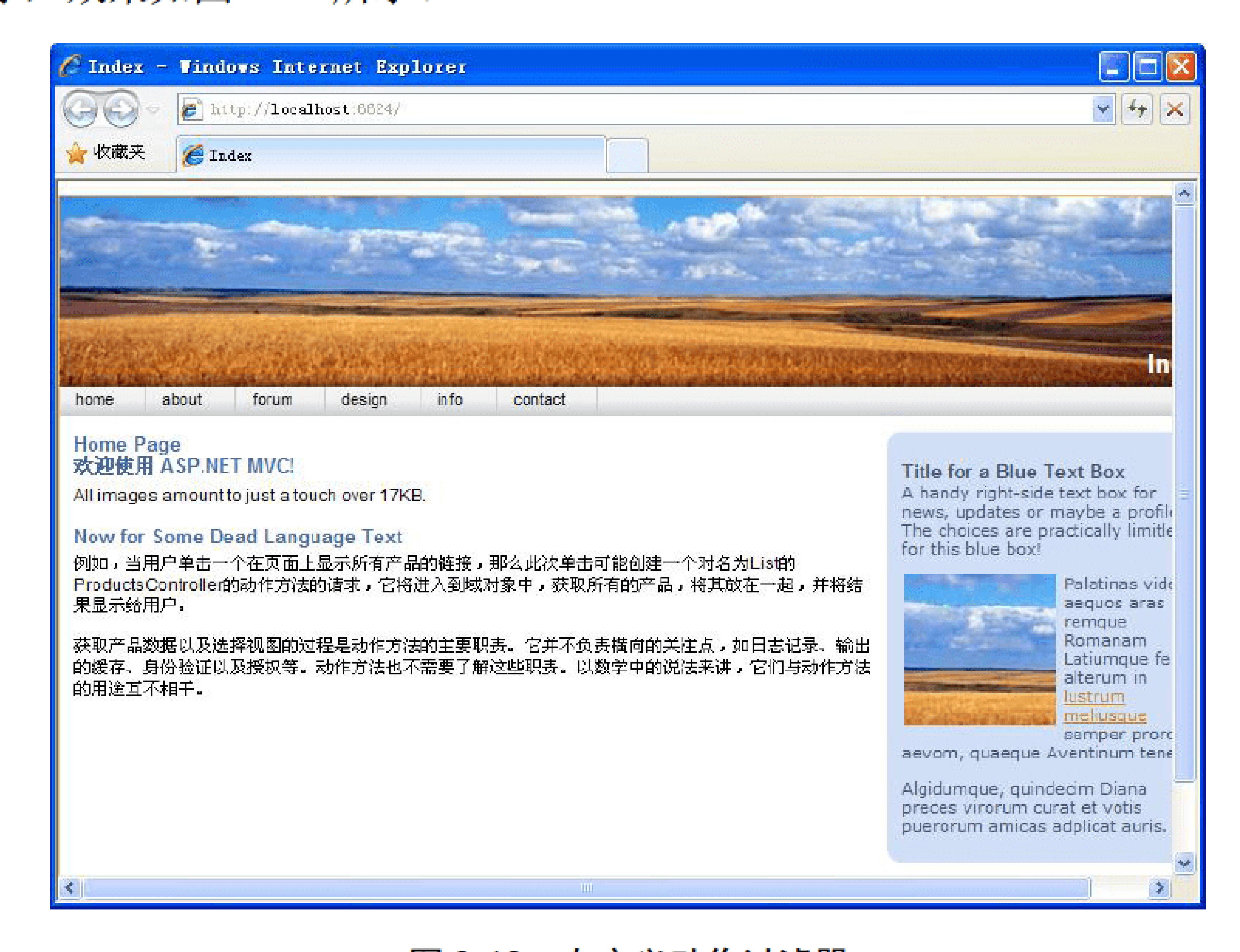


图 9-13　自定义动作过滤器

9.8.5　实例分析


源码解析

在该案例中自定义过滤器，就是自定义 TimerAttribute 类，并且该类继承自 ActionFilter-Attribute 的类，然后重写过滤器中的 4 个方法：OnActionExecuted、OnActionExecuting、OnResultExecuted 和 OnResultExecuting。

9.9　常见问题解答

9.9.1　MVC 过滤器


MVC 过滤器
网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

为什么我在使用 MVC 过滤器时，运行之后先跳到抛出异常的页面，再次运行它才跳到错误页面？

【解决办法】因为是调试模式，Debug 是开启的，如果是以 Release 方式就应该可以了。

9.9.2　使用 ASP.NET MVC 处理页面异常


如何使用 ASP.NET MVC 处理页面异常？
网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

页面出现异常，怎么处理呀？哪位高手帮忙解释一下了，谢谢！

【解决办法】在 ASP.NET MVC 中，我们可以使用 HandleErrorAttribute 特性来具体指定如何处理 Action 抛出的异常。只要某个 Action 设置了 HandleErrorAttribute 特性，那么当这个 Action 抛出异常时，MVC 将会显示 Error 视图，该视图位于~/Views/Shared 目录下。

1．设置 HandleError 属性

可以通过设置下面这些属性来更改 HandleErrorAttribute 特性的默认处理。

- ExceptionType　指定过滤器处理哪种或哪些类型的异常，如果没有指定该属性，那么过滤器将会处理所有的异常。
- View　指定发生异常时过滤器要显示的视图名称。
- Master　指定视图母版的名称。
- Order　指定过滤器应用的顺序，如果一个 Action 有多个 HandleErrorAttribute 过滤器的话。

2. 指定 Order 属性

如果某个 Action 设置了多个 HandleErrorAttribute，那么 Order 属性可以用来确定使用哪个过滤器。其值可以设置为从-1(最高优先级)到任何正整数之间的整数来标识其优先级，值越大，优先级别越低。Order 属性遵循以下规则：

- 应用到 Controller 上的过滤器将会自动应用到该 Controller 的所有 Action 上。
- 如果 Controller 和 Action 都应用了 HandleErrorAttribute，那么只要 Order 属性值相同，将会先执行 Controller 上的过滤器，而后才会执行 Action 上的过滤器。
- 对于相同 Order 属性的过滤器，其执行先后次序不定。
- 如果没有指定 Order 属性，则默认为-1，这意味着该过滤器将比其他的过滤器优先执行,除非其他过滤器指定了 Order 为-1。
- 如果有多个过滤器可适用，那么第一个可以处理该异常的过滤器会被首先调用，然后针对该异常的处理将会终结。

3. 在 View 中获取异常信息

ASP.NET MVC 框架将异常信息存储在 ViewDataDictionary 中来传递给 Error 视图，该 ViewDataDictionary 的 Model 属性即是 ExceptionContext 类的一个实例，这个 ViewData 有下面几个键。

- ActionName：目标 Action 方法的名称。
- ControllerName：目标 Controller 的名称。
- Exception：异常对象。

以上步骤可供参考。

9.10 习　　题

一、填空题

(1) AuthorizeAttribute 是包含在 ASP.NET MVC 中默认的__________过滤器。

(2) HandleErrorAttribute 是包含在 ASP.NET MVC 中的默认__________过滤器。

(3) 用来为动作方法提供输出的缓存是__________过滤器。

(4) HandleErrorInfo 类的属性包括 Action、__________和 Exception。

(5) 开发自定义动作过滤器的最简单方式是使其继承__________。

二、选择题

(1) 可以用于自定义过滤器的接口，下列选项中__________不是。

A. IAuthorizationFilter

B. IEnum

C. IResultFilter

D. IExceptionFilter

(2) OutputCacheAttribute 类的哪个属性是用于指定缓存时间的？__________

A. CacheProfile
B. Duration
C. Location
D. NoStore

(3) 下列__________是处理异常的时候需要设置的属性。

A. Action
B. Exception
C. View
D. Url

(4) 在动作调用器已经找到动作方法之后，但是在调用动作方法之前，才会调用__________。

A. OnActionExecuted
B. OnActionExecuting
C. OnResultExecuted
D. OnResultExecuting

(5) 在调用了动作方法之后，但是在执行动作结果之前，调用器将调用__________。

A. OnActionExecuted
B. OnActionExecuting
C. OnResultExecuted
D. OnResultExecuting

三、上机练习

上机练习： 编写一个将过滤器应用于 Action 的例子。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Action_Filters。

(2) 将过滤器应用 Index()动作方法，类似效果如图 9-14 所示。

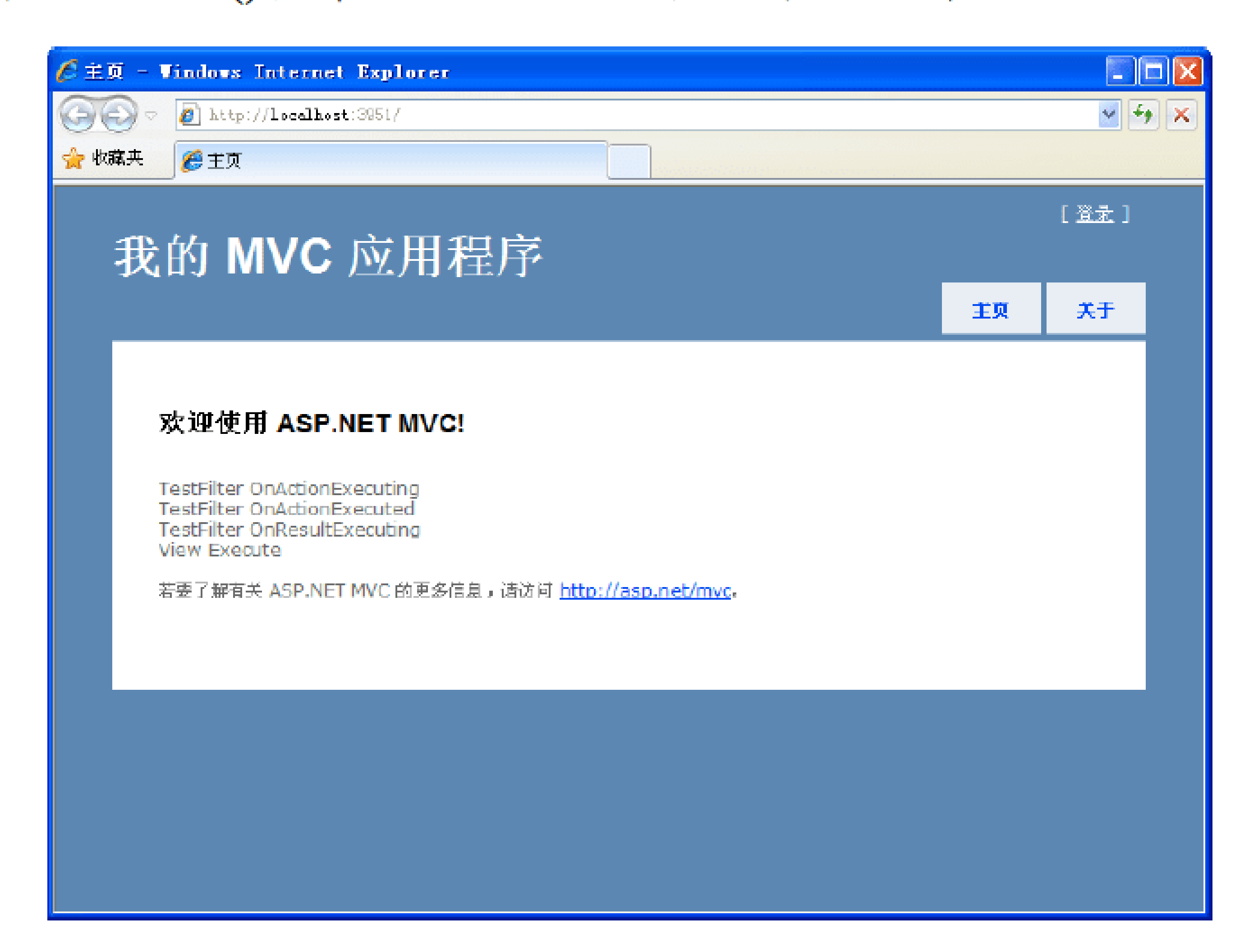


图 9-14　应用于 Action 的过滤器



第 10 章　MVC 异常处理技巧

内容摘要

在编写程序时，由于疏忽导致有些异常没有进行处理。如果将异常详细信息提示给用户会带来不安全因素，如果不提示详细信息又会给用户报告异常带来麻烦，虽然可以通过配置 customErrors 节点来实现只有管理员可以查看错误，但发现问题的时间可能会比较长。

通过全局异常处理就可以在异常发生时立即记录异常，或者直接发送邮件向管理员报告，以最快的速度发现并处理异常。

本章将从实用角度出发，为读者提供 5 种在 MVC 中处理异常的技巧。了解并掌握这些技巧，可以使构建的 MVC 项目更加安全、健壮。在实际使用时，可以综合运用其中的一种或者几种。

学习目标

- 掌握 MVC 中全局异常的处理方法
- 了解控制器异常处理方法
- 熟悉如何利用过滤器处理异常
- 了解路由异常的处理方法
- 了解动作异常的处理方法

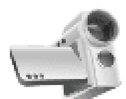
10.1　全局异常处理

设计良好的错误处理代码，可以使程序更可靠并且不容易崩溃，因为应用程序可处理这样的错误。

一般异常处理需要在程序中不断地 try、catch，并且不断地使用{}然后不断地调用。当嵌套多次之后，整个程序的层次结构会非常混乱，自己看得都有点头晕。

在诸如此类的众多处理方法中，没有比定义一个全局方法来处理更简单的事情了。

下面我们就来看一下 ASP.NET MVC 应用程序中处理全局异常的具体做法。

 **视频教学：**光盘/videos/10/10.1　全局异常处理　　　长度：9 分钟

10.1.1　基础知识——IExceptionFilter 接口

要处理全局发生的异常，就要找出 MVC 应用程序所具有的共性。

我们知道，在 MVC 应用程序中业务逻辑都放在 Controller 上，Model 负责存取数据，View 则用于显示。因此，处理全局异常实际上就是对 Controller 进行处理。

如何针对一个 MVC 应用程序中所有 Controller 添加统一异常处理方法呢？

其实很简单，MVC 在设计时早就考虑到这个问题(感叹一句，微软的东西果然一向强大)，为开发人员提供了一个方法，那就是 IExceptionFilter 接口。

IExceptionFilter 接口位于 System.Web.Mvc 命名空间，它仅包含一个 OnException 方法，其完整定义如下：

```
public interface IExceptionFilter
{
    void OnException(ExceptionContext filterContext);
}
```

也就是说，所有继承 IExceptionFilter 接口的类都必须带有一个 OnException()方法的实现。Controller 就是这样一个类，在该类中 OnException()方法是一个虚方法，其声明语法如下：

```
protected virtual void OnException(
    ExceptionContext filterContext
)
```

参数 filterContext 是 System.Web.Mvc.ExceptionContext 类型，它记录了有关当前请求和操作的信息。

该方法是每个 Controller 类都具有的，因此只需重写这个方法，便可在 Controller 中对自己的异常进行处理。

以下为 OnException()方法的一个基本型重写结构：

```
protected override void OnException(ExceptionContext filterContext)
{
    base.OnException(filterContext);
```

```
    //在这里编写异常处理语句
}
```

可以根据实际情况，在注释处编写自己的异常处理语句。下面通过实例来看看具体怎么使用吧。

10.1.2　实例描述

最近做项目时有一个想法，就是当系统内部有异常发生时，让系统自己捕获异常，然后将异常信息分类，并且把异常提示友好地展现给用户。

比如当网络中断时，提示用户网络中断；当数据库连接不上时提示数据库连接异常等。因为我不想在每个业务处理单元里写那么多 try 和 catch，所以想写一个错误处理方法来统一对异常进行处理。

以上描述相信做过开发的朋友一定不会陌生，可见全局异常的处理多么重要。

下面我们就一起来研究在 ASP.NET MVC 里如何对整个应用程序可能出现的各种异常进行统一处理。

10.1.3　实例应用

【例 10-1】全局异常处理。

(1) 打开我们使用 ASP.NET MVC 构建的 Blog 项目，在它的基础上添加功能。

(2) 为了方便日后查看与修正错误，我们需要记录异常日志。在这里我们创建一个专门记录日志的管理器类 LogManager。

LogManger 类保存在 Models 文件夹中，完整代码如下：

```
public class LogManager
{
    string LogFilePath = null;
    public LogManager(string logFilePath)
    {
        this.LogFilePath = logFilePath;
        FileInfo file = new FileInfo(logFilePath);
        if (!file.Exists)
        {
            //这里创建一个 FileStream 对象，它处于打开状态，我们需要将它关闭
            file.Create().Close();
        }
    }
    public void SaveLog(string message, DateTime writeTime)
    {
    string log = writeTime.ToString() + "  " + message;     //异常信息
    StreamWriter sw = new StreamWriter(LogFilePath, true);  //创建写入流
        sw.WriteLine(log);                                  //将信息写入文件流中
        sw.Close();                                         //关闭流
    }
}
```

可以看到，当记录异常信息时这里是保存到文件中，需要先引用 System.IO 命名空间。读

者也可以将其保存到数据库。

(3) 创建一个处理异常信息的控制器 baseExceptionController，其他控制器都继承自它。下面是它的代码：

```
using blog.Models;
namespace blog.Controllers
{
    public class baseExceptionController : Controller
    {
        protected override void OnException(ExceptionContext filterContext)
        {
            //初始化日志记录器
            LogManager logManager = new
LogManager(Server.MapPath("~/Exception.log"));
            //记录日志
            logManager.SaveLog(filterContext.Exception.Message,
DateTime.Now);
            base.OnException(filterContext);
        }
    }
}
```

这样，用于处理控制器上的全局异常类就创建完成了。baseExceptionController 控制器捕获异常以后，会将异常信息写入系统根目录下的 Exception.log 文件中。

(4) 为了使异常控制器发挥作用，我们让所有的控制器都继承自它。例如，我们的 MVC 项目中有一个 ArticleController，修改其声明为如下形式：

```
    public class ArticleController : baseExceptionController
    {
        //这里是原来的内容，保持不变
    }
```

如上述代码所示，修改后的 ArticleController 继承自 baseExceptionController。

(5) ArticleController 中的业务逻辑简单，基本上不会出错。怎么测试我们的 baseExceptionController 是否工作正确呢？

没有关系，为了测试我们直接抛出一个异常来测试项目。

对默认的 Index 动作进行小小的修改，使它被请求的时候抛出一个异常。代码如下：

```
  public ActionResult Index()
  {
      int errorCode = new Random().Next(1000, 9999);    //随机产生一个四位整数
      string errorMsg = "抱歉！您访问的资源不存在！ 错误代码： " + errorCode;
      throw new Exception(errorMsg);
      //这里是原来的内容，保持不变
  }
```

至此，我们完成了整个系统的设计。

10.1.4 运行结果

先编译再运行 Blog 项目。打开后会在浏览器中显示默认的首页，由于在该页面中并没有添加异常，所以访问正常，效果如图 10-1 所示。

图 10-1 访问首页

在浏览器的地址栏中输入 article，使其请求 Article 控制器。这时系统会产生异常，如图 10-2 所示。

因为我们是在调试项目，所以前台浏览器上能够直接看到异常信息。多刷新几次，然后从站点根目录中打开 Exception.log 文件看看记录的内容，如图 10-3 所示。



图 10-2 请求 Article 控制器

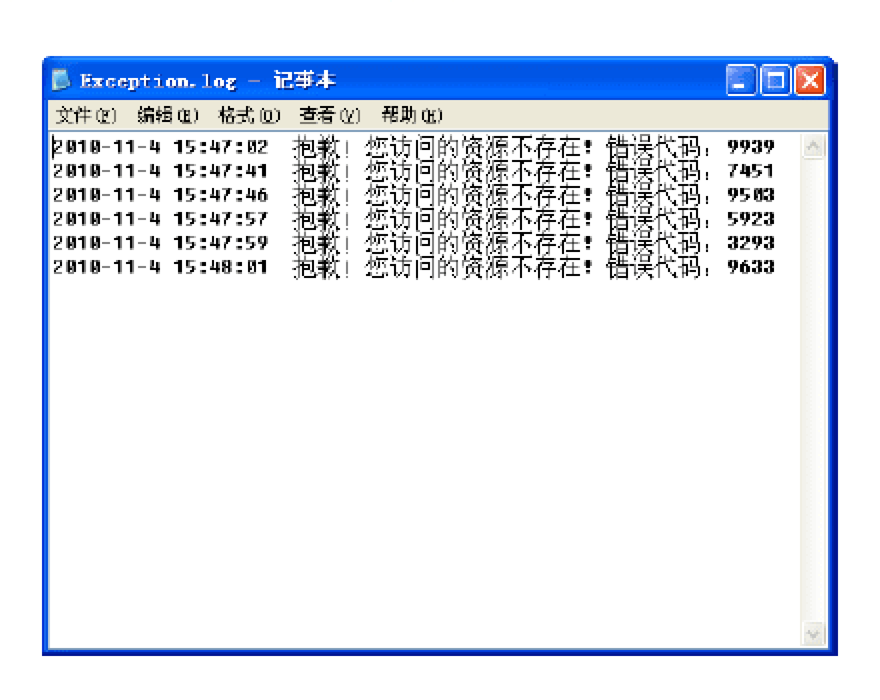


图 10-3 查看 Exception.log 文件

10.1.5 实例分析



源码解析

整体来说这个实例没有什么难点。

主要是利用 IExceptionFilter 接口的 OnException()方法，通过在控制器中重写一个作为基类，并且使用 ExceptionContext 对象的 Exception 属性来获取异常信息并记录。

之后设置让 Controller 继承这个基类，再抛出异常。

最后强调一下，这里使用 IO 对象来记录日志文件，因此必须先引用命名空间，并且在用完后及时关闭，否则会影响下次使用。

10.2 控制器异常处理

上一节通过 Controller 继承一个自定义基类，实现了对全局异常的处理。如果希望对某个控制器中的异常进行处理，应该怎么办呢？

答案很简单，在需要进行异常处理的 Controller 中重写 OnException()方法即可，因为它本身继承了 IExceptionFilter 接口。

下面就让我们跟随一个实例来看看如何实现吧，仍然以上节的 Blog 项目为基础。


视频教学：光盘/videos/10/10.2　控制器异常处理　　　长度：5 分钟

10.2.1　实例应用

【**例 10-2**】控制器异常处理。

(1) 打开我们使用 ASP.NET MVC 构建的 Blog 项目，在它的基础上添加功能。

(2) 这里要对 Article 控制器编写异常处理代码。打开 ArticleController.cs 文件，修改继承的基类为 Controller，修改后的形式如下：

```
public class ArticleController : Controller
{
    //这里是原来的内容，保持不变
}
```

(3) 添加重写的 OnException()方法。与上一节的方法不同，这里需要完整的异常处理代码，如下所示：

```
protected override void OnException(ExceptionContext filterContext)
{
    // 此处进行异常记录，可以记录到数据库或文本，也可以使用其他日志记录组件
    // 通过 filterContext.Exception 来获取这个异常
    string filePath = Server.MapPath("~/Exceptions.txt");
    StreamWriter sw = System.IO.File.AppendText(filePath);
    sw.WriteLine(filterContext.Exception.Message+"
["+DateTime.Now.ToString()+"]");
    sw.Close();
    // 执行基类中的 OnException
    base.OnException(filterContext);
    // 重定向到首页
    Response.Redirect("/");
}
```

从中可以看到，将会在站点下的 Exceptions.txt 文件中记录异常信息。之后被重定向到首页显示。

(4) 添加异常测试方法。第一个是 LogOn()方法，它将尝试返回 Account 控制器的 LogOn 视图，代码如下：

```
public ActionResult LogOn()
{
    return View("Account/LogOn");
}
```

实际运行时，上述代码将会引起一个错误。这是因为在 ASP.NET MVC 中 Controller 的 View()方法只能返回该控制器所在目录下的视图。

也就是说，这里的代码试图寻找名为 Account/LogOn 的视图，而该文件并不存在。

(5) 添加 Error1()方法，在这里直接抛出默认的异常，交给控制器去处理。代码如下：

```
public void Error1()
{
    throw new Exception();
}
```

(6) 添加 Error2()方法，这里在抛出异常时会自定义一个错误信息，然后让控制器记录这个信息，代码如下：

```
public void Error2()
{
    int errorCode = new Random().Next(1000, 9999);   //随机产生一个四位整数
    string errorMsg = "抱歉！您访问的资源不存在！错误代码：" + errorCode;
    throw new ArgumentException(errorMsg);
}
```

至此，我们完成了整个系统的设计。

10.2.2　运行结果

(1) 编译再运行 Blog 项目。

(2) 在地址栏中输入 article/error1 请求 Article 控制器的 error1 动作。此时，由于出现了异常信息会跳转到编译器，如图 10-4 所示。

(3) 继续运行，修改地址栏的值为 article/error2 请求 Article 控制器的 error2 动作，查看异常处理情况，如图 10-5 所示。

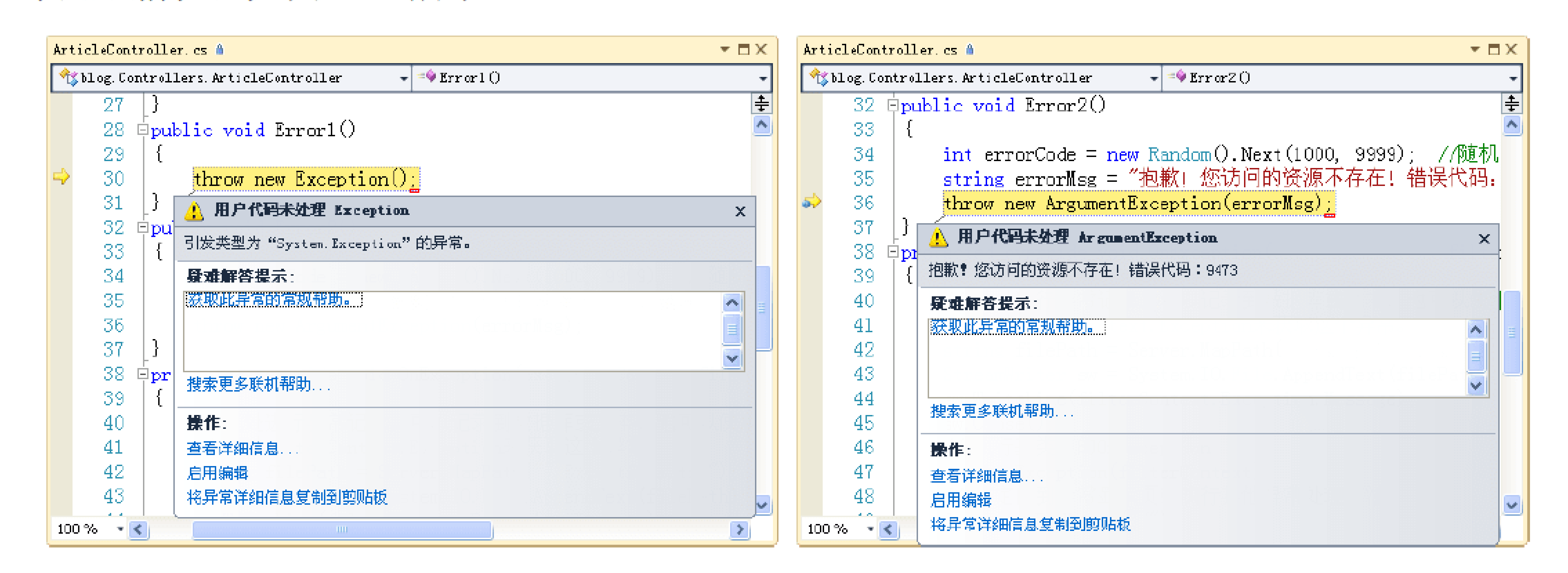


图 10-4　Error1()引发的异常　　　　图 10-5　Error2()引发的异常

(4) 修改地址栏中的值，使浏览器请求 Article 控制器的 LogOn 动作。请注意，此时不会出现前两次的错误信息，而是直接跳转到首页。

实际上，此时控制器已经将异常记录到文件。图 10-6 即为 Exceptions.txt 文件中由控制器记录的异常信息。



图 10-6　Exceptions.txt 文件中记录的异常信息

10.2.3　实例分析



源码解析

本实例与上一节实例最大的区别就是，OnException()方法的位置发生了变化。

在处理全局异常时，该方法位于一个控制器基类中，然后在应用控制器中继承它，并抛出异常。本实例则在应用控制器中重写 OnException()方法，同时抛出异常，所以应用控制器的异常不会传递到基类，而被本身的 OnException()方法捕捉并记录。这在多个控制器需要不同的异常处理方法时非常有效。

10.3　过滤器异常处理

前面的两个技巧都是在 Controller 中重写 OnException()方法，这样只要 Controller 中有异常发生，都会被该方法捕捉并处理。

但是，如果我们只想针对某个 Action 使用异常处理，那么就不能重写 Controller 的 OnException 了。有没有办法实现这种要求呢？

答案是肯定的。只需创建一个自定义的过滤器，然后在需要处理异常的 Action 上引用这个过滤器。

下面我们通过一个实例来演示具体的实现步骤。



视频教学：光盘/videos/10/10.3　过滤器异常处理　　　　长度：5 分钟

10.3.1　实例应用

【**例 10-3**】过滤器异常处理。

(1) 打开我们使用 ASP.NET MVC 构建的 Blog 项目，在它的基础上添加功能。

(2) MVC 中虽然有一个异常过滤器类 HandleErrorAttribute，但无法实现自定义的异常处理。在 Controllers 文件夹中添加一个 MyHandleErrorAttribute 类，它继承自

HandleErrorAttribute 类。

(3) 在 MyHandleErrorAttribute 类中重写过滤器中的 OnException()方法，实现对异常信息的记录。代码如下：

```
    public class MyHandleErrorAttribute : HandleErrorAttribute
    {
        public override void OnException(ExceptionContext filterContext)
        {
            base.OnException(filterContext);
            string filePath =
filterContext.HttpContext.Server.MapPath("~/MyHandleError.log");
            StreamWriter sw = System.IO.File.AppendText(filePath);
            string log = DateTime.Now.ToString() + "  " +
filterContext.Exception.Message;
            sw.WriteLine(log);
            sw.Close();
        }
    }
```

(4) 进入 ArticleController 中，在需要进行异常处理的 Action 上添加[MyHandleError]属性，以引用上一步创建的过滤器类。代码如下：

```
   [MyHandleError]
   public ActionResult ThrowMyHandle ()
   {
       int errorCode = new Random().Next(1000, 9999);   //随机产生一个四位整数
       string errorMsg = "抱歉！您访问的资源不存在！错误代码：" + errorCode;
       throw new Exception(errorMsg);
   }
```

(5) 为了演示它们的区别，下面再创建一个不带属性的 Action，代码如下：

```
    public ActionResult ThrowException()
    {
        throw new ApplicationException();
    }
```

(6) 很简单吧，到此已经完成了所有工作。现在需要保存并编译项目，看有没有什么错误。

10.3.2　运行结果

一切正常。运行项目，先在浏览器中请求不带属性的 ThrowException 动作，查看异常处理情况。

之后，再请求带有[MyHandleError]属性的 ThrowMyHandle 动作，此时会看到图 10-7 所示的异常提示，之后继续执行。

打开站点根目录下的 MyHandleError.log 文件，查看详细的异常列表。由于只有 ThrowMyHandle 动作带有自定义的过滤器属性，所以在文件中仅记录了它抛出的异常，如图 10-8 所示。

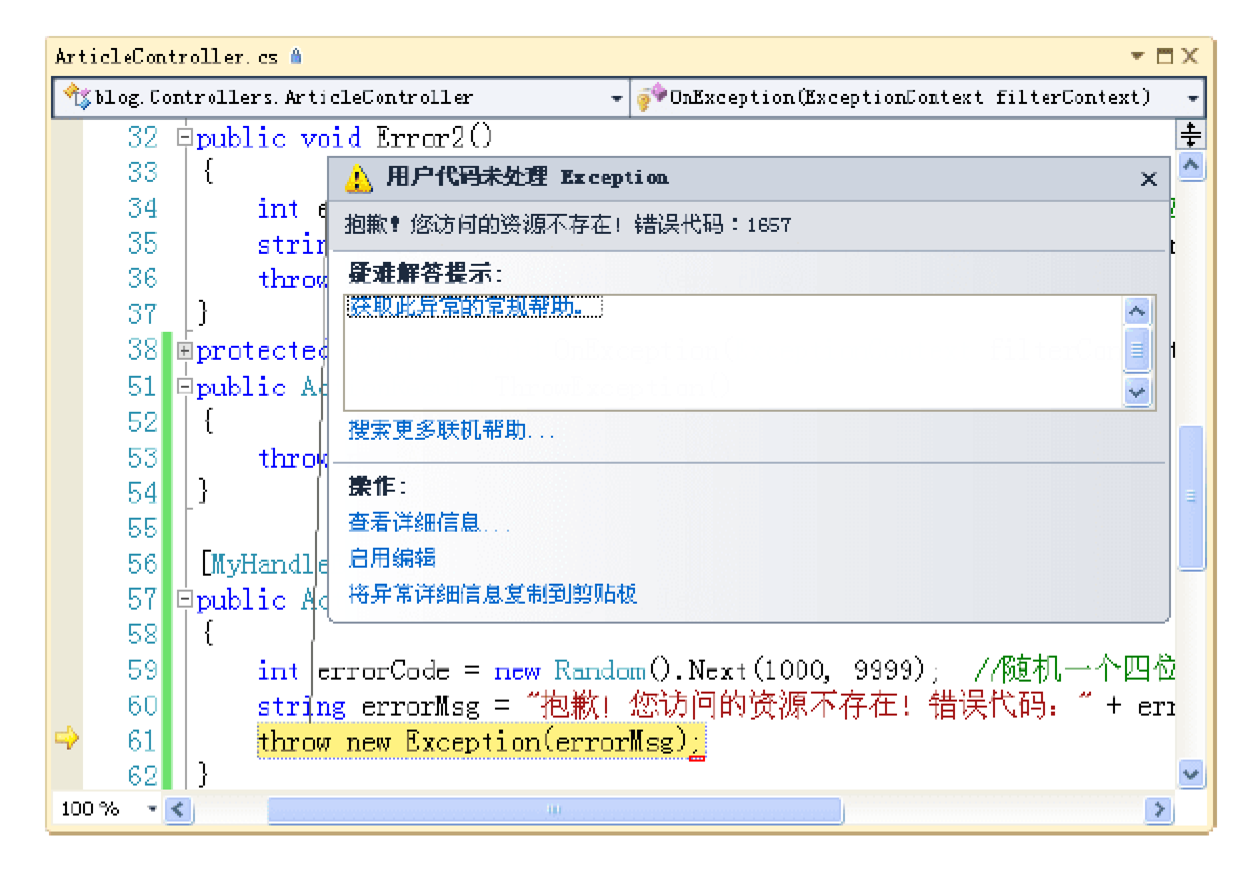


图 10-7　由动作抛出的异常提示



图 10-8　详细的异常信息列表

10.3.3　实例分析



源码解析

在本实例中演示了如何通过继承过滤器基类 HandleErrorAttribute，并重写 OnException()方法来实现自定义的异常处理。

在 ArticleController 中包含了 ThrowException()方法，该方法引发 ApplicationException 错误。但由于此方法不带任何属性，因此由系统来处理。

ThrowMyHandle()方法将引发一个新的 Exception 错误。在声明时带有[MyHandleError]属性，因此异常信息将由该属性定义的类来处理。

另外要注意，在实例中创建的过滤器类名称为 MyHandleErrorAttribute，而在添加属性时使用的名称为 MyHandleError。两者指同一个东西，但名称却不相同。

10.4　路由异常处理

我们已经知道了 ASP.NET MVC 应用程序的执行流程，在输入一个 URL 之后，会经过路由来实现对 Controller 进行映射。在路由中定义了很多映射规则表，它将对每个规则逐一进行匹配，如果正确则转到对应的 Controller 去处理。

好比银行的大堂经理(路由)，他会根据客户的需要(URL)来指定需要的步骤或者手续(路由)，像存款、取款、基金理财或者工资代扣等。

如果客户需要办理银行不具有的业务(URL 错误)。为了避免客户流失，他会解释说明并推荐与之类似的业务。

处理路由异常的工作，实际上就是当请求的 URL 找不到匹配的规则时，为它指定一个默认的匹配规则，而非显示错误。

常见的做法是在 Global.asax 文件中定义一个包罗万象的路由规则，让每一个 URL 都有请

求。下面我们将看到另一种更优雅的处理方式。

**视频教学**：光盘/videos/10/10.4 路由异常处理 长度：6 分钟

10.4.1 实例应用

【例 10-4】路由异常处理。

(1) 打开已有的 MVC 项目 Blog，在它的基础上添加功能。

(2) 打开项目根目录下的 Global.asax 文件。

(3) 对现有的路由规则进行改造，包括 Home 和 Account 控制器，代码如下：

```
 routes.MapRoute(
     "homeRoutes",                   // 路由名称
     "home/{action}/{id}",           // 带有参数的 url
     new { controller = "home", action = "index", id = UrlParameter.Optional }
     // 参数默认值
 );
 routes.MapRoute(
         "accountRoutes",            // 路由名称
         "account/{action}/{id}",    // 带有参数的 url
         new { controller = "account", action = "index", id =
UrlParameter.Optional }              // 参数默认值
     );
```

很明显，改造后的路由使它们可以更精确地进行映射操作。

(4) 下面着重介绍处理异常的路由规则。注意必须将它添加到其他主要路由规则的下方，代码如下：

```
 routes.MapRoute(
     "ErrorHandling",
     "{*str}",
     new { controller = "Error", action = "Missing" }
     );
```

因为路由规则是按照在 Global.asax 中的顺序从上往下依次执行。一旦找到匹配的，后面的就不再执行了，所以这个具有匹配所有功能的规则必须放在最后，它会转到 Error 控制器的 Missing 动作。

(5) 在当前项目中创建一个 ErrorController 类，并对其添加 Missing 动作。部分代码如下：

```
   public class ErrorController : Controller
   {
       public ActionResult Missing(string str)
       {
           ViewData["errMsg"] = "您的访问出错了,"+str + " 时间: " +
DateTime.Now.ToString();
           return View();
       }
   }
```

(6) 添加 ErrorController 对应的 Missing 视图，在这里需要显示出错信息，包括出错的路径以及返回首页的链接等。整个视图的代码如下：

```
<div class="container clearfix">
    <h2>很抱歉，您要访问的页面不存在。</h2>
    <p>
        <%=ViewData["errMsg"] %><br />
        单击这里<a href="/">返回首页</a>.
    </p>
</div>
```

至此，结束整个项目的所有工作。

10.4.2 运行结果

先编译再运行 Blog 项目。

此时会发现，由于默认的 URL 地址与路由规则中的 Home 和 Account 都不匹配，所以只好转到最后的规则，图 10-9 为此时的界面效果。

在浏览器中输入以 home/和 account/开始的 URL 则会发现运行正常。图 10-10 为输入无效 ULR 时的界面。

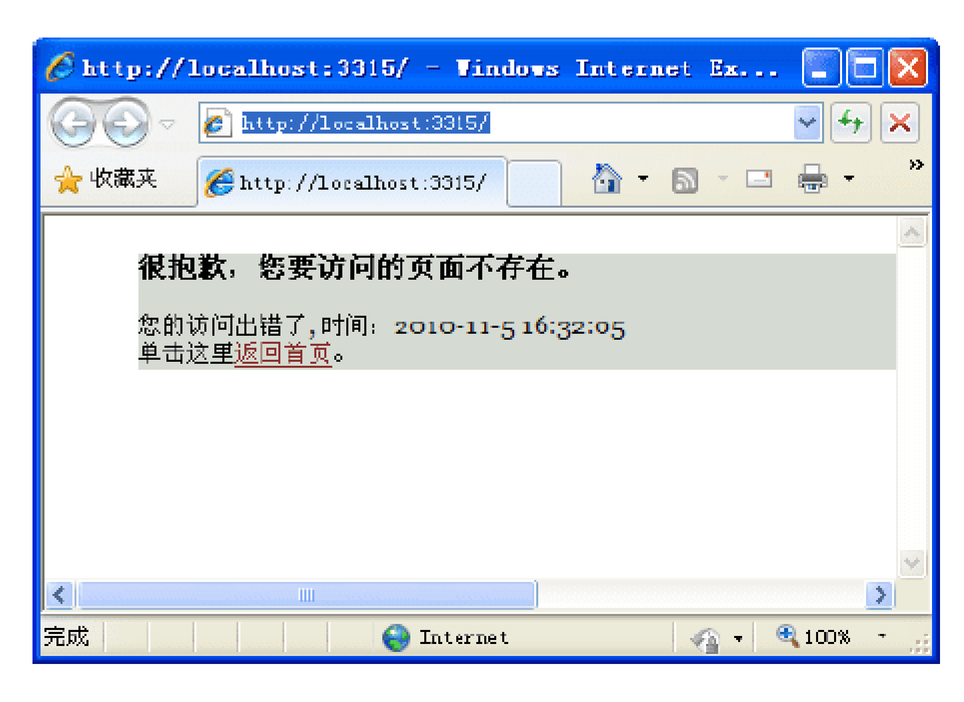


图 10-9 默认运行效果

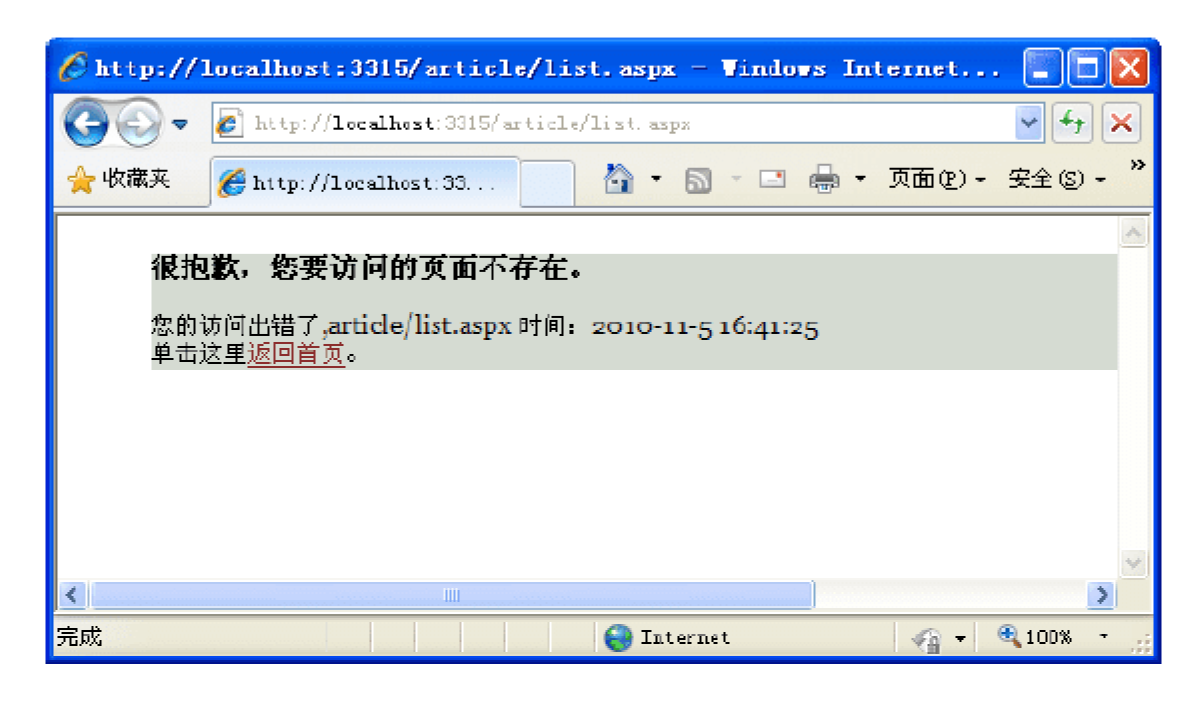


图 10-10 输入无效 URL 的效果

10.4.3 实例分析



源码解析

整个实例做下来步骤虽然不多，但是需要注意的事项可不少。

首先是在 Global.asax 文件中需要对现有的路由规则进行调整，使它们能够更准确地解析每个可能的 URL。

接下来在所有规则之后添加产生异常时的路由规则，这里需要使用通配符星号*来定义 URL。

之后，还得在项目中创建对应的控制器、动作和视图。为了在视图中显示错误信息，还必须指定一个 String 类型参数。

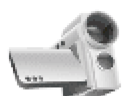
完成上述步骤之后，默认运行时就会跳转到该规则。

10.5　动作异常处理

作为一个“业务”熟悉的 Web 开发人员，除了能够修订、检测程序错误，对可能引起应用程序的异常进行捕捉外，还应该能够对用户输入无效或者错误 URL 时，做出正确的处理和响应。

因为在 ASP.NET MVC 应用程序中，一个 URL 请求会被映射到路由，之后转发到指定控制器的动作进行处理。对于不符合映射规则的 URL，用户会看到一个 HTTP 404 错误。

在这种情况下，我们希望创建一个 404 页面，告诉浏览者其请求的页面不存在或者链接错误，同时引导用户使用网站的其他页面而不是关闭窗口离开。

**视频教学：**光盘/videos/10/10.5　动作异常处理　　长度：5 分钟

10.5.1　实例应用

【例 10-5】动作异常处理。

(1) 打开一个已存在的 ASP.NET MVC 项目。

(2) 在项目的根目录下找到 Web.config 文件并打开。

(3) 要使发生 404 错误时不显示 ASP.NET 应用程序的内置错误，可以在 system.web 节点中添加一个 customErrors 节点，并设置 mode 属性为 On。

熟悉 ASP.NET 的人一定不会陌生，这里就不解释了。以下是修改后的代码：

```
<customErrors mode="On">
  <error statusCode="404" redirect="/error" />
</customErrors>
```

这样，当 ASP.NET 应用程序遇到 404 错误时，将会转到由 redirect 属性指定的 error 进行处理。



error 错误子标记可以出现多次，每出现一次便定义了一个自定义错误条件。

(4) 在当前项目中创建一个 ErrorController 类，并对默认的 Index 动作进行修改。部分代码如下：

```
public class ErrorController : Controller
{
    public ActionResult Index()
    {
        ViewData["errMsg"] = Request["aspxerrorpath"].ToString();
        //获取出错的路径
        return View();
    }
}
```

(5) 添加 ErrorController 对应的 Index 视图，在这里需要显示出错信息，包括出错的路径以及返回首页的链接等。整个视图的代码如下：

```
<h2>出错了！！</h2>
<p>
    <img src="/img/404.gif" />
    抱歉！您访问的资源"<%=ViewData["errMsg"] %>"不存在或者已经被移动!<br />
    单击这里<a href="/">返回首页</a>.
</p>
```

至此，结束整个项目的所有工作。

技术文档	customErrors 节点

customErrors 节点的格式如下所示：

```
<customerrors  defaultredirect="url"  mode="on|off|remoteonly">
   <error  statuscode="statuscode"  redirect="url"/>
 </customerrors>
```

其中，mode 属性表示是否启用或仅对远程客户端显示自定义错误。该属性是必须设置的，可选有 on、off 或者 remoteonly。

- on　指定启用自定义错误。如果没有指定 defaultRedirect，用户将看到一般性错误。
- off　指定禁用自定义错误，允许显示详细的错误信息。
- remoteonly　这是默认值。指定仅向远程客户端显示自定义错误，并向本地主机显示 ASP.NET 错误。

可选的 defaultRedirect 属性用于指定发生错误时浏览器指向默认 URL。

现在来看看 error 节点，它有两个属性，其中 Statuscode 属性用于指定发生重定向错误页时的 http 状态代码(例如 404、500 等)，Redirect 属性将向客户端展示有关该错误信息的页面。

10.5.2　运行结果

编译并运行项目，之后检查项目的原有功能是否正常。

接下来，修改浏览器的地址栏，请求一个不存在的动作。例如，请求 Products 的 list 动作，此时会显示在 Index 视图中定义的信息，如图 10-11 所示。图 10-12 为请求 Home 的 list 动作时的结果。

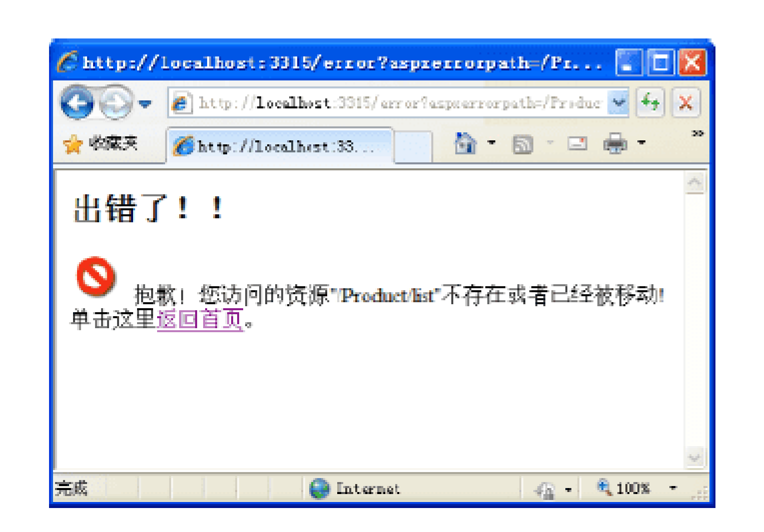


图 10-11　请求 Products 的 list 动作

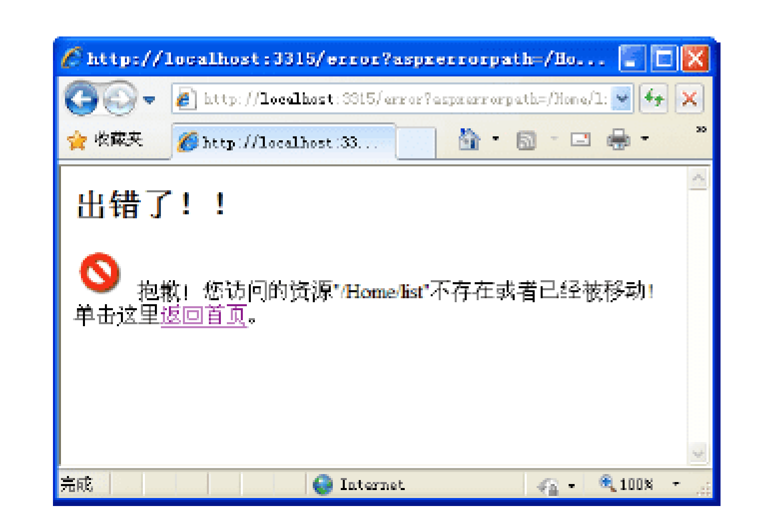


图 10-12　请求 Home 的 list 动作的结果

10.5.3　实例分析



源码解析

本实例主要利用了 Web.config 中的 customerrors 节点。在开启允许自定义错误之后，利用

error 错误子节点指定 404 错误(请求的动作或者 URL 无效)的处理方式，即转到一个自定义的控制器中。在控制器中输出了请求的无效动作以及错误提示。

这样，当用户在运行项目时，不会因为误输入错误的 URL 而导致整个应用程序崩溃，而是显示比较人性化的错误提示。

10.6　常见问题解答

10.6.1　global.asax 中的错误处理



global.asax 中的错误处理

网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html l

大家好，这里有个问题纠结很长时间了。

我有一个 MVC 项目，想在 global.asax 里集中处理应用程序中的错误。我创建了一个默认显示错误信息的视图，位于~/Views/Shared/Error.aspx。

问题来了，为了在不使用控制器 HandleError 属性的情况下，也不指定 customErrors 节点来实现。我在 global.asax 文件中添加了如下的代码，它只能在部分环境下运行正常。却无法正常运行在带集成管道模式的 IIS7 下，这是什么原因？

- Visual Studio 2010 自带的开发环境 Web 浏览器。
- 基于脚本映射的 IIS6 模式。
- 标准的 IIS7 带脚本映射环境下。

```
    public class MvcApplication : System.Web.HttpApplication {
        public static void RegisterRoutes(RouteCollection routes) {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                "Default",                                  // Route name
                "{controller}/{action}/{id}",               // URL with parameters
                new { controller = "Home", action = "Index", id = "" }
                                                           // Parameter defaults
            );
        }
        protected void Application Start() {
            RegisterRoutes(RouteTable.Routes);
        }
        protected void Application Error() {
            HttpContext ctx = HttpContext.Current;
            KeyValuePair<string, object> error = new KeyValuePair<string,
object>("ErrorMessage", ctx.Server.GetLastError().ToString());
            ctx.Response.Clear();
            RequestContext rc =
((MvcHandler)ctx.CurrentHandler).RequestContext;
            string controllerName =
rc.RouteData.GetRequiredString("controller");
            IControllerFactory factory =
ControllerBuilder.Current.GetControllerFactory();
```

```
            IController controller = factory.CreateController(rc,
controllerName);
            ControllerContext cc = new ControllerContext(rc,
(ControllerBase)controller);
            ViewResult viewResult = new ViewResult { ViewName = "Error" };
            viewResult.ViewData.Add(error);
            viewResult.ExecuteResult(cc);
            ctx.Response.End();
        }
```

【解决办法】估计问题出在 Application_Error()处理错误的代码中。这是因为，如果你想使用自定义的错误视图而非默认的 ASP.NET 黄屏提示，那么在不同的 IIS 下处理方式略有不同。

依据我的经验，global.asax 文件是运行在 IIS6 和 IIS7 集成管道模式的默认应用程序池中。下面是我测试通过的代码，要注意 ctx.Server.ClearError()的位置。

```
  protected void Application_Error() {
      HttpContext ctx = HttpContext.Current;
      KeyValuePair<string, object> error = new KeyValuePair<string,
object>("ErrorMessage", ctx.Server.GetLastError().ToString());
      ctx.Response.Clear();
      RequestContext rc = ((MvcHandler)ctx.CurrentHandler).RequestContext;
      string controllerName = rc.RouteData.GetRequiredString("controller");
      IControllerFactory factory =
ControllerBuilder.Current.GetControllerFactory();
      IController controller = factory.CreateController(rc, controllerName);
      ControllerContext cc = new ControllerContext(rc,
(ControllerBase)controller);
      ViewResult viewResult = new ViewResult { ViewName = "Error" };
      viewResult.ViewData.Add(error);
      viewResult.ExecuteResult(cc);
      ctx.Server.ClearError();
      //ctx.Response.End();
    }
```

10.6.2 ASP.NET MVC 中的异常处理



global.asax 中的异常处理

网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

我使用 ASP.NET MVC 建立了一个默认的错误视图。

```
 public partial class Error : ViewPageBase {
    //处理错误的视图
 }
```

之后又创建了一个 Controller 和一个 Show 动作，代码如下：

```
public ActionResult Show(string id) {
      return View("Error");
    }
```

如何实现？当 ID 不为空时(像 404、502 等)，我想显示一个消息以说明错误。如果 ID 为空，则显示通用的信息。

当控制器下的动作有 HandleError 属性时，实例的 ID 可以为空。这个说法正确吗？

我知道这个错误有一个类型 HandleErrorInfo 对象具有 ActionName、ControllerName 和异常属性。

但是，怎样得到这个对象，以便在 Show 动作中将日志保存到数据库中呢？

还有一个问题，我是否可以移动 Error 控制器下视图文件夹下的 Error.aspx 文件呢？

【解决办法】HandleErrorAttribute 就是以这种方式工作的：如果你希望在某个动作抛出异常时显示“错误”视图，则可以添加[HandleError]属性。标准的 HandleErrorAttribute 仅会显示一些视图，但是你希望的是重定向错误。

对于楼主你这样的情况，最好的解决方案是实施自己的错误处理属性。这种做法并不难实现，只需创建 FilterAttribute 类的派生类和继承 IExceptionFilter 接口即可。

剩下的就是重写 onException 方法，就像下面给出的代码：

```
    public class RedirectOnErrorAttribute : FilterAttribute, IExceptionFilter
    {
        public void OnException(ExceptionContext filterContext)
        {
            filterContext.Result = new RedirectToRouteResult(new
RouteValueDictionary(
                new { action = "Show", controller = "Error",
                      id = new HttpException(null,
filterContext.Exception).GetHttpCode().ToString(),
                      exceptionAction =
(string)filterContext.RouteData.Values["action"],
                      exceptionController =
(string)filterContext.RouteData.Values["controller"]
                }));
            filterContext.ExceptionHandled = true;
        }
    }
```

除此之外，在 Web.Config 中你还可以做一些额外的配置。

```
    <customErrors mode ="On" defaultRedirect ="/Error/Generic">
      <error statusCode ="400" redirect ="/Error/Http/400"/>
      <error statusCode ="401" redirect ="/Error/Http/401"/>
      <error statusCode ="403" redirect ="/Error/Http/403"/>
      <error statusCode ="404" redirect ="/Error/Http/404"/>
      <error statusCode ="408" redirect ="/Error/Http/408"/>
      <error statusCode ="502" redirect ="/Error/Http/502"/>
      <error statusCode ="503" redirect ="/Error/Http/503"/>
      <error statusCode ="504" redirect ="/Error/Http/504"/>
    </customErrors>
```

怎么样，明白了吧！我想用不了多少时间你就可以弄懂，要相信自己。

10.6.3 为什么 Controller 的 HandleError 属性不会覆盖 Action 的 HandleError 属性



为什么 Controller 的 HandleError 属性不会覆盖 Action 的 HandleError 属性？
网络课堂：http://bbs.itzcn.com/thread-2976-1-1.html

先说一下问题出现的场景吧。

我的 MVC 项目中有很多 Controller 和 Action，我希望所有 Action 的异常都由默认的 Error.aspx 视图来显示。

另外还有一些 Action，我想让它们在自定义的 ErrorDialog.ascx 视图中显示。编写了下面的代码：

```
[Authorize]
[HandleError]
public class SkillsMatrixController : Controller
{
    public ActionResult Personal() { return View(); }

    [AcceptVerbs("GET")]
    [HandleError(View="ErrorDialog")]
    public PartialViewResult MarketDialog() { return PartialView(); }
}
```

由上面的代码可以看到，它应该可以达到我预期的效果。

现在的问题是，当我做完以上操作后，所有的错误均由 Error.aspx 来处理了。我以为 Action 上的 HandleError 属性会覆盖 Controller 上的 HandleError 属性。事实却不是这样，请教各位高手，在 MVC 框架下怎么达到这种效果呢？

我试过下面的改良代码，但仍然没有效果。

```
[Authorize]
public class SkillsMatrixController : Controller
{
    [HandleError]
    public ActionResult Personal() { return View(); }

    [AcceptVerbs("GET")]
    [HandleError(View="ErrorDialog")]
    public PartialViewResult MarketDialog() { return PartialView(); }
}
```

【解决方法】在 MVC 的 Controller 上使用 HandleError 属性时，有一个 Order 选项，当某个方法有多个 HandleErrorAttribute 筛选器时，它可以指定筛选器的应用顺序。在没有显式地指定该值时，默认都是-1，因此最终都会转到带有 HandleError 属性的 Controller 来处理。

希望上面的说明能够帮助你理解。

10.7 习　　题

一、填空题

(1) 为了能够自定义异常处理方法，需要继承__________接口。

(2) OnException()方法是一个__________方法，它的 filterContext 参数是一个__________类型。

(3) 为了使用户能够自定义错误处理页面，需要在 Web.config 中添加__________节点。

(4) error 节点的__________属性用于指定发生像 404 或者 500 这样的 HTTP 状态代码。

二、选择题

(1) IExceptionFilter 接口位于__________命名空间下。

A. System.Web.Mvc

B. System.Web.Exception

C. System.Web.MVC.Exception

D. System.Web.MVC.IException

(2) 下列哪个方法是由 IExceptionFilter 接口提供的？__________

A. ThrowException()方法

B. OnException()方法

C. HandleException()方法

D. ExceptionFilter()方法

(3) 下面关于 IExceptionFilter 接口的描述，错误的是？__________

A. IExceptionFilter 接口位于 System.Web.Mvc 命名空间

B. Controller 类继承 IExceptionFilter 接口

C. IExceptionFilter 接口有一个 OnException()方法

D. IExceptionFilter 接口有一个 OnExceptionFilter()方法

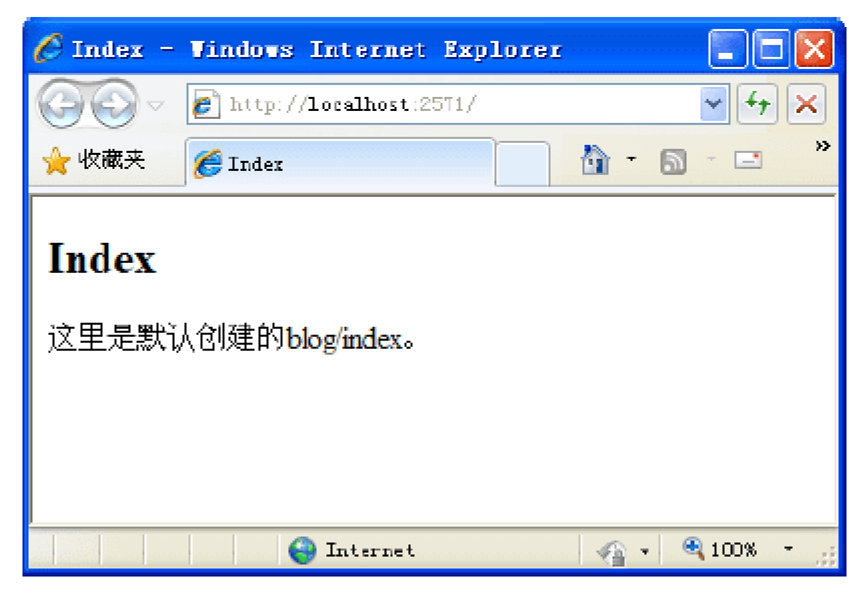
三、上机练习

上机练习 1： 构建完善的异常处理机制。

本章针对 MVC 项目中处理异常的各种方法进行了介绍。本次练习要求读者根据下面的描述，自己构建一个完善的异常处理解决方案。

假设，这里有一个非常简单的 Blog 网站。在这个网站中有两个栏目：文章和相册。现在要用 MVC 来开发这个网站，主要实现如下功能：

- 输入网站地址后，默认显示 blog/index 视图，如图 10-13 所示。
- 文章栏目的访问地址为 blog/article。
- 相册栏目的访问地址为 blog/photo。
- 除上述正常的访问地址外，访问其他地址时均显示访问出错，如图 10-14 所示。



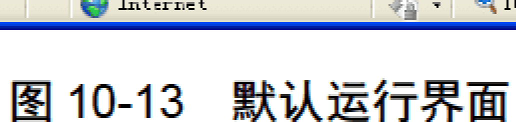


图 10-13　默认运行界面

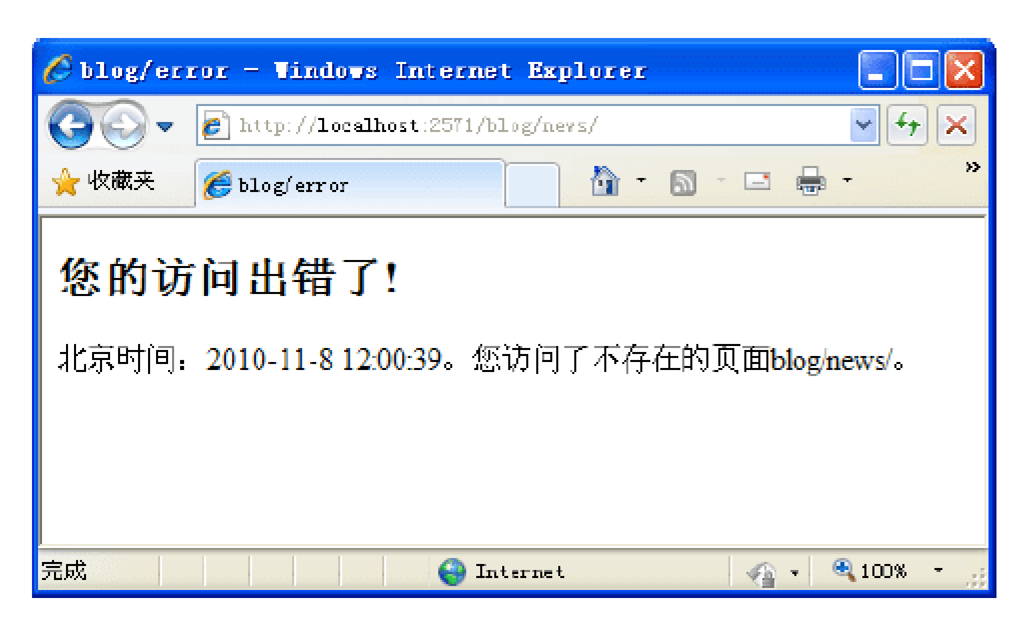


图 10-14　访问出错界面

上机练习 2： 使用 OnException()方法处理异常。

本次练习的目的是使读者更加熟练地掌握 OnException()处理异常的方法。

(1) 打开一个 MVC 项目，或者从默认的 MVC 示例项目中开始。

(2) 打开一个 Controller，重写 OnException()方法。

(3) 编写代码，当出现错误时将信息记录到外部文件中。

(4) 创建一个测试方法 Error1()，编写代码以定义异常信息并抛出。

(5) 编译并运行项目，之后调用 Error1()方法，抛出异常时的提示信息，如图 10-15 所示。

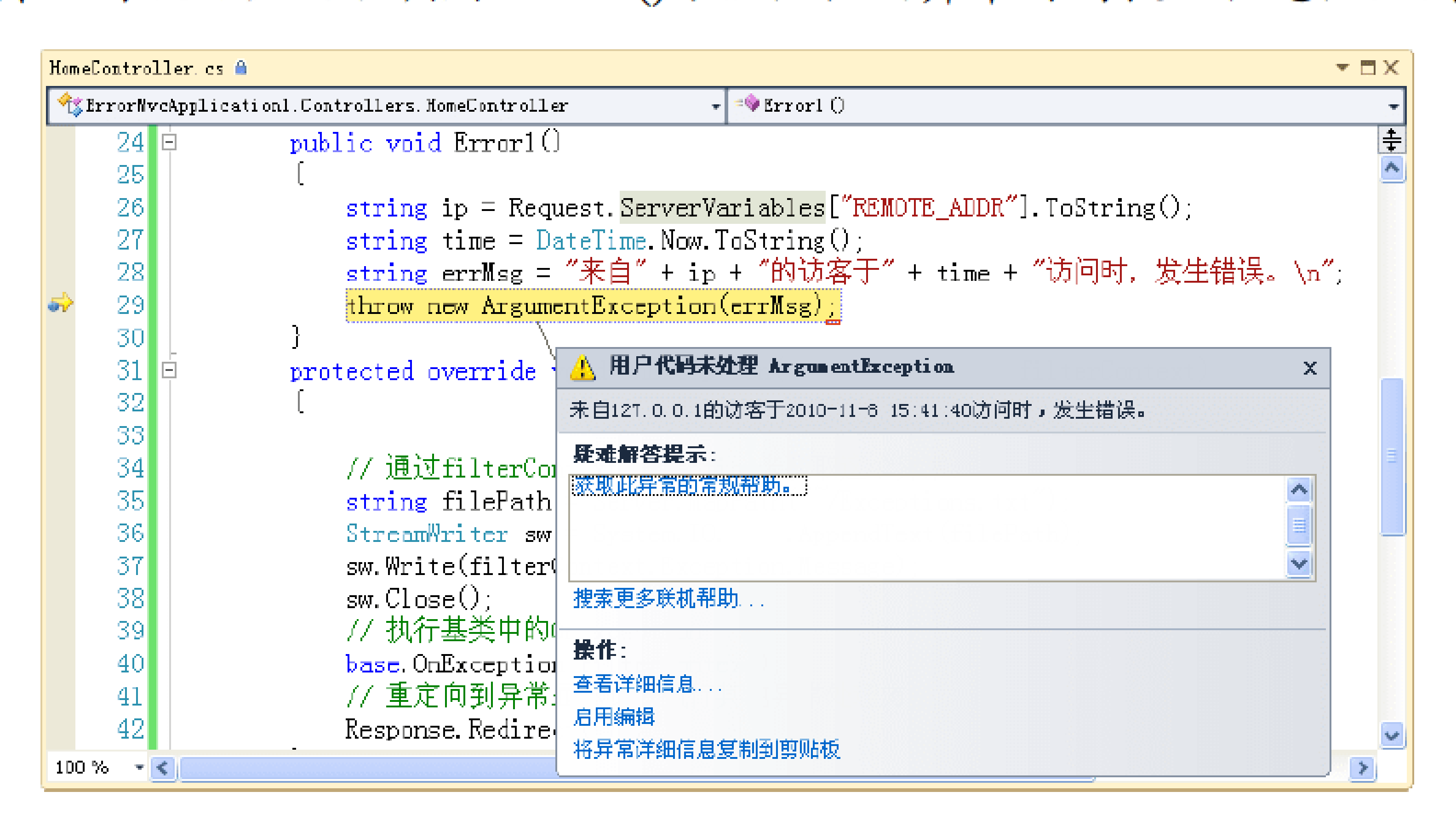
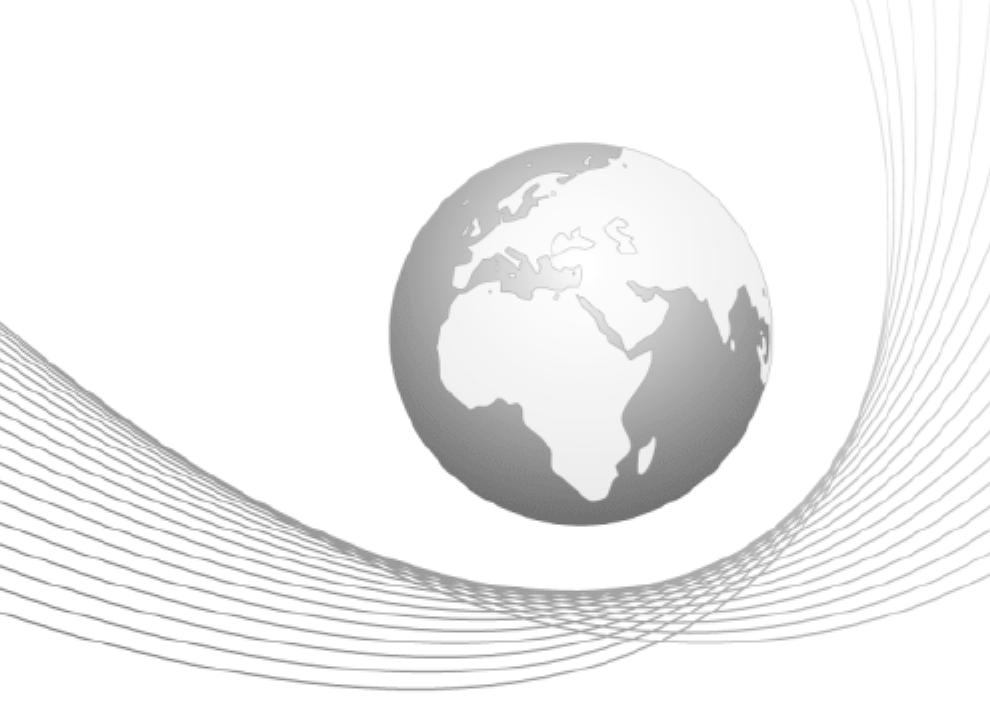


图 10-15　抛出异常时的提示



第 11 章　MVC 中 jQuery 的应用

内容摘要

jQuery 是一个优秀的开源 JavaScript 库，其设计理念就是“写得少，做得多”。它的语法简单易学，而且具有强大的跨平台特性，可以兼容多种流行的浏览器，使其在众多优秀的 JavaScript 库中脱颖而出，独树一帜，赢得了众多 Web 开发人员的拥护和信赖。

在创建 MVC 项目时，jQuery 也作为默认的 JavaScript 库被支持，更可以看到 jQuery 的优秀。本章中将着重讲解在 View 中使用 jQuery 来处理前台显示逻辑和布局，例如获取表单信息，突出显示页面某一元素，控制显示列表以及显示效果等。

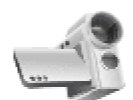
学习目标

- 掌握如何在视图中引入 jQuery
- 熟悉 jQuery 的基本选择器和过滤选择器
- 掌握使用 jQuery 对页面元素的搜索
- 掌握如何定位指定的页面元素
- 熟悉 jQuery 获取表单数据的方法
- 掌握读取 CSS 的方法
- 掌握获取 CSS 的方法
- 了解对视图元素应用 jQuery 动画的方法
- 熟悉 jQuery UI 中日历的使用
- 熟悉 jQuery UI 中对话框的使用

11.1　利用$()获取页面元素信息

对一个熟练的 Web 编程人员来说，在编写 JS(JavaScript 的简称)库代码时候，经常会和$符号打交道。

无论 prototype 还是 DW(DreamWeaver 的缩写)都会使用$来代替频繁的 document.getElementById()操作。jQuery 也会这样做，但是它的功能远非如此。我们研究一下 jQuery 的代码，就会发现它的魅力。

**视频教学：**光盘/videos/11/11.1　MVC 中 jQuery 的应用(1)　　长度：20 分钟
MVC 中 jQuery 的应用实例(2)　　长度：10 分钟

11.1.1　基础知识——jQuery 选择器

通俗地讲，选择器(Selector)就是一个表示特殊语意的字符串。

选择器是 jQuery 的一个亮点，jQuery 选择器可分为基本选择器和过滤选择器两类，并且可以配合使用，组合成一个选择器字符串。主要区别是过滤选择器是指定条件从前面匹配的内容中筛选。过滤选择器也可以单独使用，表示从全部的即*中筛选。

1．基本选择器

基本选择器包含 CSS 选择器、层级选择器和表单域选择器。下面就来介绍这些基本选择器的用法。

1) CSS 选择器

jQuery 借用了一套 CSS 选择器，共有 5 种，即标签选择器、ID 选择器、类选择器、通用选择器和群组选择器。

表 11-1 中列出了这 5 种 CSS 选择器的语法说明及示例。

表 11-1　CSS 选择器

名　称	语法说明	示　例
$("element")	选择 HTML 页面中已有的标签元素，参数 element 表示待查找的 HTML 标记名	$("div")
$ ("id")	获取某个具有 ID 属性的元素，参数 id 表示待查找元素的 id 属性值	$("#user")
$("class")	获取某个具有 class 属性的元素，参数 class 指定应用于带选择元素的类名	$(".item")
$("*")	匹配所有元素	$("*")
$("selector1,selectorN")	选择所有指定选择器组合的结果，参数可以为有效的任何选择器	$("div,span,p.styleClass")

2) 层级选择器

在 HTML 文档中，每个元素总是处在 DOM 节点树上的某个位置，文档层次结构中元素之间总是存在着某种层级关系，例如父级与子级的关系等。在 jQuery 中，可以使用层级选择器来获取这类相关的元素。

表 11-2 中列出了这些层级选择器的语法说明及示例。

表 11-2　层级选择器

名　称	语法说明	示　例
$("parent>child")	在给定父元素下查找所有子元素。参数 parent 和 child 是任何有效的选择器；child 是第一个选择器的子元素，用于筛选子元素。两个参数之间用>分隔	$("form > input")
$("ancestor descendant")	在给定祖先元素下匹配所有后代元素。参数 ancestor 和 descendant 是任何有效的选择器，后代元素可能是 ancestor 元素的儿子、孙子或重孙等。两个参数之间用空格分开	$("form input")
$("prev+next")	匹配所有紧接在 prev 元素后的 next 元素。prev 和 next 表示任何有效选择器，两者用+分隔	$("prev+next")
$("prev~siblings")	参数 prev 和 siblings 是任何有效的选择器，用于筛选 prev 后面的所有同辈元素。两者之间用波浪线~分隔	$("div~input")

3) 表单域选择器

表单域就是指网页中的 input、textarea、select 和 button 元素，其中 input 元素可以具有各种各样的 type 属性值，例如 text、password、radio 以及 checkbox 等。jQuery 提供了一组选择器，专门用于从文档中选择表单域。这些选择器均以冒号:开头，其中大多数都可以独立使用。

表 11-3 中列出了这些表单域选择器的语法说明。

表 11-3　表单域选择器

名　称	语法说明
$(":input")	选择所有 input、textarea、select 和 button 元素
$(":text")	选择所有单行文本框(____)
$(":password")	选择所有密码框( )
$(":radio")	选择所有单选按钮(☐)
$(":checkbox")	选择所有复选框(☐)
$(":file")	选择所有文件域(<input type="file"/>)
$(":image")	选择所有图像域(<input type="image"/>)
$(":hidden")	选择所有不可见元素以及隐藏域
$(":submit")	选择所有提交按钮(<input type="submit"/>和<button>…</button>)
$(":reset")	选择所有重置按钮(<input type="reset"/>)

2．过滤选择器

使用基本选择器在文档中查询时，jQuery 对象通常会包含一组 DOM 元素。在实际应用中，

往往还要根据特定条件从获取的元素集合中筛选出一部分 DOM 元素。

在这种情况下，可以在基本选择器的基础上添加过滤选择器来完成查询任务。根据具体情况，在过滤选择器中可以使用元素的索引值、内容、属性、子元素位置、表单域属性以及可见性作为筛选条件。

1) 简单过滤

简单过滤选择器主要根据索引值对元素进行筛选，它们均以冒号:开头，并且要与另外一个选择器配合使用。

表 11-4 中列出了这些简单过滤选择器的语法说明及示例。

表 11-4　简单过滤选择器

名　称	语法说明	示　例
$("selector:first")	对当前 jQuery 集合进行过滤并选择第一个匹配元素	$("td:first")
$("selector:last")	对当前 jQuery 集合进行筛选并选择最后一个匹配元素	$("td:last")
$("selector:odd")	选择索引为奇数(从 0 开始计数)的所有元素	$("td:odd")
$("selector:even")	选择索引为偶数(从 0 开始计数)的所有元素	$("selector:even")
$("selector:eq(index)")	从匹配的集合中选择索引等于给定值的元素。参数 index 指定元素在 selector 集合中的索引值(从 0 开始计数)	$("li:eq(1)")
$("selector:gt(index)")	参数 index 指定元素在 selector 集合中的索引值(从 0 开始计数)，只有索引大于此值的元素才会包含在查询结果中	$("li:gt(0)")
$("selector:lt(index)")	参数 index 是一个非负整数，用于指定元素在 selector 集合中的索引值(从 0 开始计数)，只有索引小于此值的元素才会包含在查询结果中	$("li:lt(0)")
$("selector1:not(selector2)")	从匹配的集合中去除所有与给定选择器匹配的元素。参数 selector1 和 selector2 均表示任何有效的选择器	$("td:not(:first,:last)")
$(":header")	选择所有诸如 h1、h2 和 h3 之类的标题元素	$(":h1")
$("selector:animated")	选择所有正在执行动画效果的元素	$("div:not(:animated)")

2) 内容过滤

在 HTML 文档中，元素的内容可以是文本或子元素。如果将某个选择器或内容过滤选择器配合使用，就可以从查询到的元素中进一步筛选出具有给定文本或子元素的元素。

表 11-5 中列出了这些内容过滤选择器的语法说明及示例。

表 11-5　内容过滤选择器

名　称	语法说明	示　例
$("selector:contains(text)")	选择包含给定文本的所有元素。参数 selector 表示任何有效的选择器，text 指定要查找的文本(引号是可选的)	$("p:contains('hi')")

续表

名　称	语法说明	示　例
$("selector1:has(selector2)")	选择含有给定子元素的元素。selector1 和 selector2 均为任何有效的选择器。如果 selector1 元素至少包含一个与 selector2 匹配的元素，则该元素将包含在查询结果中	$("li:has(p)")
$("selector:empty")	选择不包含子元素或文本的所有空元素。参数 selector 是任何有效的选择器，selector 集合中不包含子元素或文本的所有空元素将被包含在查询结果中	$("td:empty")
$("selector:parent")	选择包含子元素或文本的元素，与:empty 选择器的作用相反	$("td:parent")

3) 属性过滤

在 jQuery 中，除了直接使用 id 和 class 属性作为选择器之外，还可以根据各种属性(如 title 等)对由选择器查询到的元素进行过滤。属性过滤选择器包含在中括号[]里，而不是以冒号开头。通过使用“选择器[属性过滤选择器]”语法格式，可以根据是否包含指定属性或根据属性值从查询到的元素中进行筛选。

表 11-6 中列出了这些属性过滤选择器的语法说明及示例。

表 11-6　属性过滤选择器

名　称	语法说明	示　例
$("selector[attribute]")	选择包含给定属性的所有元素，参数 attribute 表示属性名	$("div[id]")
$("selector[attribute=value]")	选择给定属性等于特定值的所有元素，参数 attribute 表示属性名，value 表示属性值	$("input[name=accept]")
$("selector[attribute*=value]")	选择指定属性值包含给定子字符串的所有元素。参数用于指定要查找的元素；参数 attribute 为属性名，参数 value 为属性值	$("input[name*='news']")
$("selector[attribute~=value]")	选择指定属性值中包含给定单词(由空格分隔)的元素	$("input[name~='news']")
$("selector[attribute!=value]")	选择不包含指定属性，或者包含指定属性但该属性不等于某个值的所有元素	$("input[name!='newsletter']")
$("selector[attribute^=value]")	选择给定属性是以某特定值开始的所有元素	$("input[name^='news']")
$("selector[attribute$=value]")	选择指定属性是以某个给定值结尾的所有元素	$("input[name$='news']")
$("selector[selector1][selectorN]")	选择同时满足多个条件的所有元素	$("input[id][name$='man']")

4) 子元素过滤

子元素过滤选择器必须与某个选择器配合使用。首先使用这个选择器进行查询，由此得到一个(父)元素数组，然后按照子元素过滤选择器指定的索引值或规则，为该数组中的每个(父)元素进一步筛选出部分子元素。

表 11-7 中列出了这些子元素过滤选择器的语法说明及示例。

表 11-7　子元素过滤选择器

名　称	语法说明	示　例
$("selector:first-child")	选择其父级的第一个子元素的所有元素	$("ul:first-child")
$("selector:last-child")	选择其父级的最后一个子元素的所有元素	$("ul:last-child")
$("selector:nth-child(index)")	选择父元素下的第 N 个子元素或奇偶元素	$("ul li:nth-child(4)")
$("selector:only-child")	如果某个元素是父元素中唯一的子元素，那将会被匹配，否则将不会被匹配	$("ul li:only-child")

5) 表单域属性过滤

表单内包含各种各样的表单域，使用表单域属性选择器可以轻松地获取已被选中的单选按钮、复选框以及列表项，也可以根据是否可用从文档中查找表单域。

表 11-8 中列出了这些表单域过滤选择器的语法说明及示例。

表 11-8　表单域过滤选择器

名　称	语法说明	示　例
$("selector:checked")	选择所有被选中的表单域。参数 selector 用于指定要查找的元素类型，可以是 input、radio 或 checkbox	$("input:checked")
$("selector:enabled")	选择所有可用的表单域	$("input: enabled ")
$("selector:disabled")	选择所有被禁用的表单域	$("input:disabled")
$("selector:selected")	从列表框里选择所有选中的 option 元素	$("select option:selected")

6) 可见性过滤

如果某元素及其父元素在文档中占有空间，则认为该元素为可见。反之，如果某元素及其父元素在文档中不占用空间，则认为该元素为不可见。

表 11-9 中列出了这些可见性过滤选择器的语法说明及示例。

表 11-9　可见性过滤选择器

名　称	语法说明	示　例
$("selector:hidden")	选择所有不可见元素	$("tr:hidden")
$("selector:visible")	选择所有可见元素	$("tr:visible")

11.1.2　实例描述

之前坚持认为使用原生的 JavaScript 才是王道，那种直接书写 document.getElementById 的

方式让我感到无尽的喜悦。应当说 JavaScript 本身十分优雅并且有一种神秘，让前端开发人员倍感煎熬的不是 JavaScript 诡异的方式，更多的应该是不同浏览器对 DOM 及 JavaScript 的解析。

你应当还记得 if(navigator.userAgent.toLower().indexOf('msie'))吧？一方面使用 jQuery 你完全不必考虑 textContent 与 innerText，另一方面 jQuery 强大之处便是它的选择器。相信你只要有 CSS 的基础，就能很快使用 jQuery 写出强大而简洁的语句。

古语讲“工欲善其事，必先利其器”，所以我的建议是一定要使用 jQuery。现在它已经成为我做项目的唯一 JavaScript 库。

下面将向大家介绍如何利用 jQuery 获取页面元素信息。

11.1.3 实例应用

【例 11-1】利用$()获取页面元素信息。

(1) 新建一个 MVC 项目，命名为 jqueryMvcApplication。

(2) 添加一个名称为 Member 的 Controller。

(3) 在 MemberController 中新建一个名称为 Index 的 Action，并返回默认视图。

(4) 新建一个母版页，并添加名称为 TitleContent 和 MainContent 的内容页。



后面的操作都是在本实例中创建的 MVC 项目上进行扩展，使用的 Controller 也都是 Member。

(5) 进入 Index 视图文件，添加如下代码，引入所需的 jQuery 类库。

```
    <script src="../../Scripts/jquery-1.4.1.min.js"
type="text/javascript"></script>
```

(6) 设计实例的布局，添加标题、表格、表单元素和提交按钮等。部分代码如下：

```
    <h2>注册成为本站的会员</h2>
    <fieldset>
        <legend>完善下面的信息 </legend>
        <table border="0" width="500" id="MemberArea">
         <thead>
             <tr><td colspan="2"><h3>用户注册</h3></td>  </tr>
         </thead>
         <tbody>
             <tr>
                 <td class="td left">登录名： </td>
                 <td><%=Html.TextBox("loginName") %></td>
             </tr>
             <tr>
                 <td class="td left">密码： </td>
                 <td><%=Html.Password("password") %></td>
             </tr>
             <tr>
                 <td class="td left"> 确认密码： </td>
                 <td> <%=Html.Password("password2") %></td>
             </tr>
             <tr>
                 <td class="td_left"> 性别： </td>
```

```
                <td> <%=Html.RadioButton("sex", true, new { style = "border:0;
width:30px;" })%>男  <%=Html.RadioButton("sex", false, new { style = "border:0;
width:30px;" })%>女
                </td>
            </tr>
            <tr>
                <td class="td left">已婚： </td>
           <td> <%=Html.CheckBox("married", false, new { style = "border:0;
width:30px;" })%> </td>
            </tr>
            <tr>
                <td class="td left"> 安全邮箱：  </td>
                <td> <%=Html.TextBox("email") %>输入你常用的邮箱(找回密码使用)</td>
            </tr>
            <tr>
                <td class="td left"> 联系电话： </td>
                <td> <%=Html.TextBox("phone") %>输入你常用的联系电话(或手机)</td>
            </tr>
            <tr>
        <td colspan="2" align="center"> <br /> <input id="Button1" type="button"
value="确定" /> </td>
            </tr>
        </tbody>
    </table>
 </fieldset>
```

(7) 编写 jQuery 代码，首先编写一个函数，它在页面加载完成后执行。

```
    <script language="javascript" type="text/javascript">
        $(document).ready(function () {
            //页面加载完成后执行
        });
    </script>
```

(8) 使用选择器对页面的外观进行修改，包括改变表格间隔行的背景色，为表单名称添加背景色以及设置标题的字体大小。代码如下：

```
     $("#MemberArea tbody tr:even").css("backgroundColor", "#E6F4FE");
    //分组选择器
     $(".td left").css({ "backgroundColor": "#12eb87", "color": "#000" });
    //类选择器
   $("legend").css("fontsize", 14);                           //标签选择器
```

(9) 编写代码实现单击【确定】按钮后获取输入的信息并弹出显示。部分实现代码如下：

```
     $("#Button1").click(function () {                             //ID选择器
        msg = "你好，用户：" + $("#loginName").val() + ",\n"
        + "请记好你的密码：" + $("#password").val() + ",\n"
        + "邮箱：" + $("#email").val() + ",\n"
        + "电话：" + $("#phone").val() + "。";
        alert(msg);                                       //弹出对话框
     });
```

在这段代码中，msg 变量保存了使用 jQuery 获取的用户名、密码、邮箱以及电话。Alert()是 JavaScript 的内置函数，可以弹出一个显示字符串的对话框。

(10) 打开项目的 Global.asax 文件，设置默认 Controller 为 Member，默认 Action 为 Index。至此，我们完成了整个系统的设计。

11.1.4　运行结果

先编译再运行本节的 jqueryMvcApplication 项目。

由于修改了 Global.asax 文件，打开后浏览器中显示的默认视图为 member/index。在该页面中会显示预先设计的内容。

在表单中输入注册信息，单击【确定】按钮将看到结果，如图 11-1 所示。

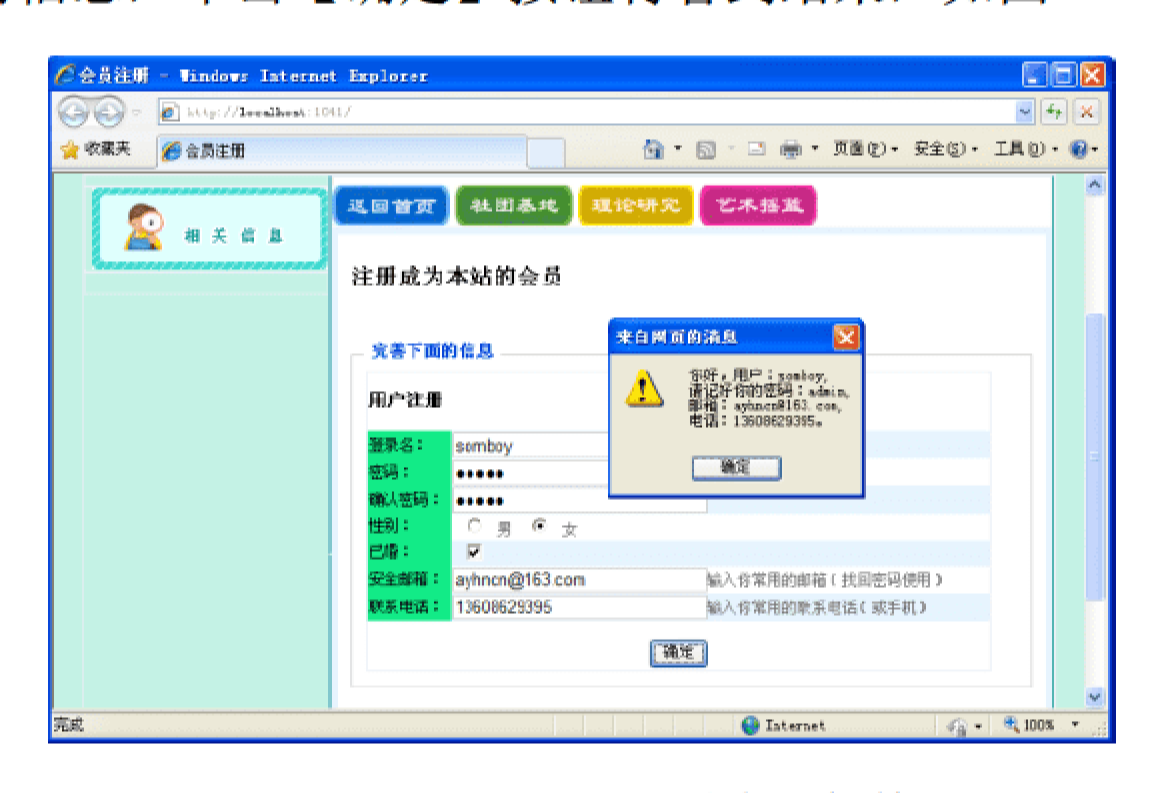


图 11-1　获取页面元素信息运行效果

11.1.5　实例分析



源码解析

由于 jQuery 会作为 MVC 的默认 JavaScript 类库被放在 Scripts 目录下，因此在 View 中很容易将其引入文件，但是要注意路径以及文件名必须保持一致。

```
<script src="../../Scripts/jquery-1.4.1.min.js" type="text/
 javascript"></script>
```

接下来，本实例利用 jQuery 中的选择器修改页面元素的显示样式，以及获取输入的数据并弹出显示。

另外，还需要注意的是 jQuery 代码的编写位置是在页面加载完成后执行，即在 jQuery 的 $(document).ready()函数内。

11.2　遍历所有的相同元素

开发人员在做项目的时候，为了使页面整齐和美观，通常会对页面元素设置样式，其中最常用的就是 jQuery。当需要对页面的某个或者某些元素进行设置时，就要先搜索这些元素，然后对它们的样式表进行设置。

jQuery 提供了很多操作指定元素的方法，例如搜索、过滤、串联和筛选等。在这里我们来看看如何在页面中搜索具有同一层次的页面元素。

视频教学：光盘/videos/11/11.2　遍历所有的相同元素　　长度：8 分钟

11.2.1　基础知识——搜索同辈元素

在常规的 DOM 编程中，可使用 previousSibling 和 nextSibling 属性来检索与当前元素相邻的同辈元素。jQuery 提供了类似的方法来搜索当前元素的同辈元素，而且功能更为强大。

表 11-10 中列出了这些搜索同辈元素方法的语法说明及示例。

表 11-10　搜索同辈元素方法

名　称	语法说明	示　例
next([selector])	获取紧跟在每个匹配元素之后的单个同辈元素，根据需要还可以指定一个选择器对同辈元素进行筛选	$("p").next("p").css("color","#FCF");
nextAll([selector])	搜索跟在每个匹配元素之后的所有同辈元素	$("p").nextAll().css("color","blue");
nextUntil([selector])	获取跟在每个匹配元素后面的同辈元素直至匹配给定选择器的元素(但不包括该元素)	$("#div1").nextUntil("div").css("border","1px solid red");
prev([selector])	搜索紧邻每个匹配元素前面的单个同辈元素	$("#div2").prev("span").css("color","blue");
prevAll([selector])	搜索当前元素之前所有的同辈元素	$("#div2").prevAll("span").css("color","blue");
prevUntil([selector])	搜索当前元素之前所有的同辈元素，直到遇到匹配的那个元素为止	$("#div2").prevUntil("input").css("color","red");
siblings([selector])	搜索每个匹配元素的所有同辈元素，还可以指定一个选择器对这些同辈元素进行筛选	$("div").siblings().css("color","red");

11.2.2　实例描述

前段日子的空闲时间比较充裕，利用这些时间对学校的网站进行了局部优化。主要是在显示列表内容的时候利用 jQuery 来改善其表现方式(像校内新闻栏目、教学文件栏目等)，修改后的效果非常不错。

终于有时间整理一下了，下面就将此方法公布出来，希望对正在研究的朋友有所帮助。

11.2.3　实例应用

【例 11-2】遍历所有的相同元素。

(1) 在 MemberController 中添加名称为 List 的 Action。

(2) 为 List 添加 View，并在返回默认的视图文件中应用上一节创建的母版页。

(3) 制作一个列表文件，这里使用 ul 标记来实现。部分代码如下：

```
<ul>
    <li>2010 年 11 月份教学工作计划<span>[ 2010-10-4 ]</span></li>
    <li id="t1">第十一周媒体 0901 专业实训课表<span>[ 2010-10-4 ]</span></li>
    <li>2010 年下学期计算机信息技术第十一周实训课表
<span>[ 2010-10-4 ]</span></li>
    <li>2010 级 IBM 高职软件技术专业人才培养方案<span>[ 2010-9-10 ]</span></li>
    <li id="t2">多媒体技术专业 2010 级人才精细化培养方案
<span>[ 2010-9-10 ]</span></li>
    <li>计算机网络技术专业人才精细化培养方案<span>[ 2010-9-10 ]</span></li>
    <li>计算机信息管理专业人才精细化培养方案<span>[ 2010-9-10 ]</span></li>
    <li id="t3">2010 下学期我系教学任务概况<span>[ 2010-9-1 ]</span></li>
    <li>信息工程系 2010 年下学期 09 级重修工作安排<span>[ 2010-9-1 ]</span></li>
    <li>ACCP 班 Y2 补考考场安排表<span>[ 2010-9-1 ]</span></li>
</ul>
```

(4) 引入 jQuery 的类库文件 jquery-1.4.1.min.js。

```
<script src="../../Scripts/jquery-1.4.1.min.js"
type="text/javascript"></script>
```

(5) 编写页面加载完成的 jQuery 代码，如下所示：

```
<script language="javascript" type="text/javascript">
    $(document).ready(function () {
    })
</script>
```

(6) 在 ready()函数内为 t1、t2 和 t3 进行搜索并设置 CSS 样式。

```
    //给 t1 下一个同辈元素设置字体颜色
    $("#t1").next().css("color", "#F0F");
    //给 t1 元素后面的所有同辈元素设置边框
    $("#t1").nextAll().css("border", "1px solid #F00");
    //给 t2 元素前面的一个同辈元素设置字体颜色
    $("#t2").prev().css("color", "#99c");
    //给 t2 元素前面的所有同辈元素设置背景
    $("#t2").prevAll().css("background", "#FCF");
    //给所有 t3 的同辈元素设置字体颜色
    $("#t3").siblings().css("color", "#99F");
```

11.2.4　运行结果

至此已完成对实例的修改。运行页面输入 member/list 请求并查看效果，图 11-2 为应用之前的效果。

现在，添加 jQuery 代码再看一看应用之后的效果，如图 11-3 所示。

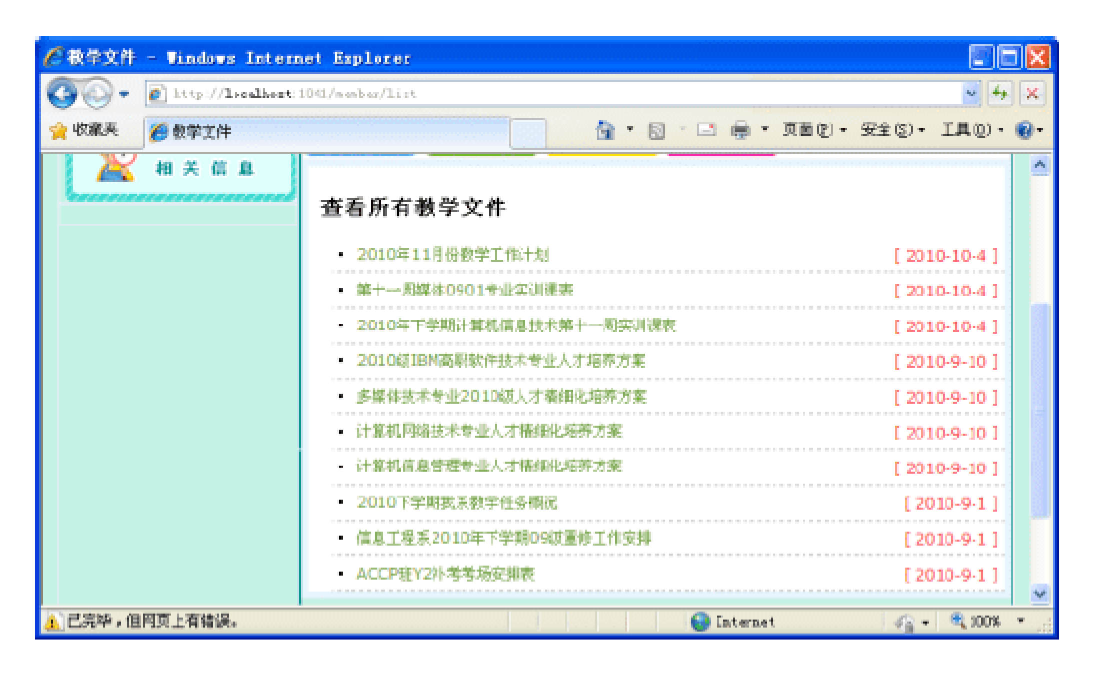


图 11-2　应用之前的效果

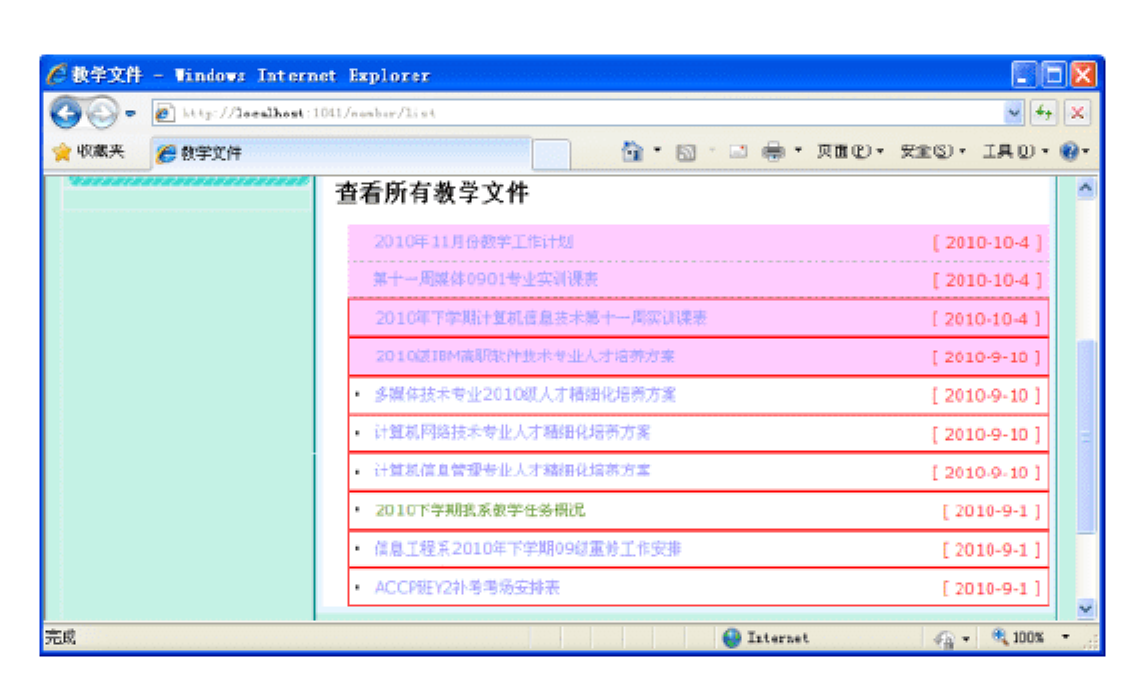


图 11-3　应用之后的效果

11.2.5　实例分析



源码解析

在本实例的 List 视图文件中仅用了一个 ul 标记，包含的多个 li 标记均位于同一层中。通过设置 3 个 id 属性，指定搜索时的开始位置。

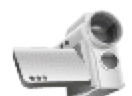
从运行效果的两幅图中可以看出，应用之前所有列表均显示相同样式。应用之后，整个列表被标识为很多小块，每一块都具有单独的显示样式。

从而可以更加清晰地区分它们的状态。例如，用 t1 标识最新发布的，t2 标识已经下载过的，t3 标识已过期的等。

11.3　突出显示图片

平时网上冲浪，我们会被一些显示效果非常特别的东西所吸引。例如，在一个普通的新闻列表中，可以使用加粗或者其他颜色来增加权重。在一个图片列表中，随着鼠标的移动，选中的某张图片将会突出显示。

类似实现手段都只有一个目的，即希望被更多地关注。下面我们就来学习一种最简单的实现方式。



视频教学：光盘/videos/11/11.3　突出显示图片　　长度：6 分钟

11.3.1　基础知识——eq()方法

jQuery 可以对搜索到的页面元素进行过滤操作，也就是对被查找元素集合进行筛选。其中最主要的是 eq()方法，其语法格式如下：

```
eq(index)
```

其中，参数 index 表示索引值(从 0 开始)。执行后可以获取第 index 个元素。

假设有如下 HTML 代码：

```
<table width="200" border="1">
  <tr>
    <td> </td>
    <td> </td>
  </tr>
  <tr>
    <td> </td>
    <td> </td>
  </tr>
</table>
```

现在要筛选出第 4 个单元格，并将它的背景色设置为#FCF，代码如下：

```
$("td").eq(3).css("background","#FCF");
```

11.3.2　实例描述

如何在一些图片列表中突出显示某一张图片呢？

相信做 Web 网站开发的朋友都遇到过类似的问题，我就曾在一个项目中为此功能而苦战了两天还没弄正确。

后来，有幸得到一位专家的妙方才得以实现。下面给出解决与实现方法，有兴趣的朋友共同研究一下。

11.3.3　实例应用

【例 11-3】突出显示图片。

(1) 在 MemberController 中添加名称为 Photo 的 Action。

(2) 为 Photo 添加 View，并在返回默认的视图文件中应用母版页。

(3) 引入 jQuery 的类库文件 jquery-1.4.1.min.js。

(4) 制作一个图片列表，并包含一个标题。最终布局代码如下：

```
<div id="photos">
    <h1>最受欢迎的课程列表</h1>
    <img src="/images/yuwen1.jpg" />
    <img src="/images/yuwen2.jpg" />
    <img src="/images/gushi.jpg"  title="故事大王"/>
    <img src="/images/shuxue.jpg" />
</div>
```

可以看到，在 ID 为 photos 的容器中包含了 4 个用 img 标记显示的图片。其中，第 3 张将是后面用代码来实现突出显示的。

(5) 编写 jQuery 代码，实现预期的效果。

```
<script language="javascript" type="text/javascript">
    $(document).ready(function () {
        //使第 3 张图片突出显示
       $("#photos img").eq(2).css("border-color", "#F00");
    })
</script>
```

(6) 保存对 View 和 Controller 的修改，重新编译项目。

11.3.4 运行结果

运行项目，在浏览器中输入 member/photo 请求查看效果。页面打开后会看到在 4 张图片中只有第 3 张使用了红色边框显示，把鼠标移到上面时还会显示提示，如图 11-4 所示。



图 11-4 突出显示第 3 张图片效果

11.3.5 实例分析



源码解析

整个实例的制作过程比较简单，唯一需要注意的是 eq()方法的使用。

在应用 eq()方法之前可以用 jQuery 选择器从页面中匹配所需的页面元素，而且这些元素应该是一个集合。另外，eq()方法传递的索引值是从 0 开始。

11.4 获取调查表单的数据

最开始的网页都是静态网页，即基于 HTML 表单控件来进行开发的。像文本框、按钮和下拉列表之类的常用表单控件，都可以用来满足用户的基本需求。

本节我们就来创建一个静态的调查表单，然后利用 jQuery 获取 View 中的表单数据。



视频教学：光盘/videos/11/11.4 获取调查表单的数据　　长度：11 分钟

11.4.1 基础知识——val()方法

本节我们将介绍一个专门用于操作表单元素的方法——val()方法。val()方法可以获取和设置表单元素的值，包括文本框、下拉列表框、单选按钮以及复选框等。

1. 获取元素值

当 val()方法不带参数时，则返回第 1 个匹配元素的值。如果是可多选的元素，则返回一个数组，其中包含选中的每个值。语法格式如下：

```
val()
```

假设有如下一段的 HTML 代码：

```
昵称：<input type="text" id="nickname" value="itzcn" /><br />
兴趣：<select id="intest" multiple="multiple">
      <option selected="selected">上网</option>
      <option selected="selected">音乐</option>
      <option>游戏</option>
    </select>
```

要获取“姓名”文本框和“爱好”列表框中的元素值，可用如下的代码：

```
    var strName=$("#nickname").val();        //结果为：itzcn
    var aryFonds=$("#intest").val();         //结果为：["上网", "音乐" ]
```

当然，也可以结合:selected 和:checked 选择器来获取元素的值。例如：

```
    $("select option:selected").val();              //获取多选列表框的值
    $("select").val();                               //获取多选列表框的值(简洁方式)
    $("input:checkbox:checked").val();              //获取一个选中复选框的值
    $("input:radio[name=intest]:checked").val();    //获取一个单选按钮组的值
```

2．设置元素值

当在 val()方法中传递一个字符串或者数组作为参数时，此参数将用来设置匹配集合中每个元素的值。语法格式如下：

```
val(value)
```

例如，将上面“姓名”文本框的值设置为 hi jQuery 的代码如下：

```
    $("#nickname").val("hi jQuery");                 //通过一个字符串为元素赋值
```

将上面“爱好”列表框中的所有值选中，可用如下的代码：

```
    $("#intest").val(["上网","音乐","游戏"]);         //通过一个数组为元素赋值
```

3．根据索引设置元素值

val()方法也允许将一个函数作为参数，其语法格式如下：

```
val(function(index, value))
```

function(index, value)函数接收两个参数：index 为元素在集合中的索引位置，text 为当前元素的 text 值。在此函数可以用 this 来指向当前元素，返回一个要设置的元素值。

例如，在所有复选框元素的值前添加 chk_前缀，代码如下：

```
   $("input:checkbox").val(function(index,value){
     return "chk " +$(this).val();
   });
```

11.4.2　实例描述

下面通过实例来看一下如何使用 val()方法读取和设置表单元素的值。首先需要一个包含表单元素的表单，在这里我们综合了各种常见的表单元素，包括文本框、密码框、复选框、单选按钮、多选列表框、下拉列表框以及多行文本框和按钮等。

11.4.3 实例应用

【例 11-4】获取调查表单的数据。

(1) 在 MemberController 中添加名称为 Diaocha 的 Action。

(2) 为 Diaocha 添加 View，并在返回默认的视图文件中应用母版页。

(3) 引入 jQuery 的类库文件 jquery-1.4.1.min.js。

(4) 制作调查表单，并将它们放在一个 table 中。下面是实例所需的表单代码。

```
    <table width="470" align="center" cellpadding="5" cellspacing="1" >
        <tr height=20>
          <th width="445" >用 户 调 查 表 单</th>
        </tr>
        <tr>
          <td >用户名:<input  type="text" id="nameIpt" value="游客" size="10">
            密码: <input  type="password" id="passIpt" value="123456"
size="10"></td>
        </tr>
        <tr>
          <td>常去网站:
            <input type="checkbox" name="chkWebSite"
value="csdn">CSDN  
            <input type="checkbox" name="chkWebSite"
value="msdn">MSDN  
            <input type="checkbox" name="chkWebSite"  value="xinwen">新闻网
  
            <input type="checkbox" name="chkWebSite"  value="ccw">计世网
  
            <input type="checkbox" name="chkWebSite"  value="jiaocheng">教程网
</td>
        </tr>
        <tr>
          <td>默认搜索:
            <input type="radio" name="rdSearch" value="baidu">baidu  
            <input type="radio" name="rdSearch"
value="google">google  
            <input type="radio" name="rdSearch" value="yahoo">yahoo
  
            <input type="radio" name="rdSearch" value="msn">MSN</td>
        </tr>
        <tr>
          <td valign="top">掌握技能:
            <select name="select" multiple="multiple" id="select1">
              <option>前台 Web 设计</option>
              <option>网站优化与推广</option>
              <option>ASP.NET 网站开发</option>
              <option>js 客户端编程</option>
              <option>XML 应用</option>
              <option>DIV+CSS</option>
            </select>
            最想了解的技术:
            <select name="select2" id="select2">
```

```
            <option>jQuery 开发</option>
            <option>.NET 框架</option>
            <option>设计模式</option>
            <option>sql 数据库</option>
            <option>网络通信</option>
          </select></td>
      </tr>
      <tr>
        <td valign="top">简单描述:<br>
          <textarea name="textarea" cols="50" rows="2" id="more">网
站:www.itzcn.com</textarea></td>
      </tr>
      <tr>
        <td><input type="submit" id="btnSubmit" value="提交">
          <input type="reset"  id="btnReset" value="重置"></td>
      </tr>
    </table>
```

(5) 在页面的合适位置添加一个表格用来显示结果，具体代码如下：

```
    <table width="100%"  border="0" align="center">
      <tr>
        <th height="20">结 果 显 示 区 域</th>
      </tr>
      <tr>
        <td id="res"></td>
      </tr>
    </table>
```

ID 为 res 的单元格用来显示用户在表单中选择的结果。

(6) 编写 jQuery 代码，利用 val()方法设置表单元素的赋值。代码如下：

```
<script language="javascript" type="text/javascript">
$(document).ready(function(){
    //设置密码框的值
    $("#passIpt").val("hellojquery");
    //设置复选框的值
    $("input:checkbox[name='chkWebSite']").val(["msdn","csdn"]);
    //设置单选按钮的值
    $("input:radio[name='rdSearch']").val(["baidu"]);
    //设置多选列表框的值
    $("#select1").val(["前台 Web 设计","ASP.NET 网站开发"]);
});
</script>
```

此时运行页面将会发现有很多表单元素会被选中，说明在上述代码中对表单元素值的修改已经生效。

(7) 为表单中“提交”按钮的单击事件编写代码，使其被单击后获取当前所有表单元素的值，并显示到指定的区域内。完整实现代码如下：

```
    $("#btnSubmit").click(function(){                    //单击“提交”按钮后执行
        var nameStr=$("#nameIpt").val();                 //获取“姓名”文本框的值
        var passStr=$("#passIpt").val();                 //获取“密码”文本框的值
        var webSites=Array();                            //获取“网站”复选框的值
        $("input:checkbox[name='chkWebSite']:checked").each(function(){
```

```
            webSites[webSites.length]=$(this).val();
        });
        //获取“搜索”单选按钮的值
        var searchs=$(":radio[name='rdSearch']:checked").val();
        var skill=$("#select1").val();              //获取“技能”多选列表框的值
        var tech=$("#select2").val();               //获取“技术”下拉列表框的值
        var text=$("#more").val();                  //获取“描述”多行文本框的值
        var btnName=$("#btnSubmit").val();          //获取当前按钮的值
        var res=btnName+"按钮被按下，各个选项的结果为：<br/>"
               +"<u>姓名：</u>"+nameStr+"，<u>密码：</u>"+passStr
               +"<br/><u>常去网站：</u>"+webSites+"<br/><u>默认搜索：
</u>"+searchs
               +"<br/><u>掌握技能：</u>"+skill+"<br/><u>想了解技术：</u>"+tech
               +"<br/><u>个人描述：</u>"+text;      //将各个值进行格式化
        $("#res").html("您的选择结果如下<hr/>"+res);  //显示所有以上获取的值
    });
```

至此，整个实例制作完成。

11.4.4　运行结果

运行项目，在浏览器中输入 member/diaocha 请求查看效果。在页面中对表单元素进行改变后，单击【提交】按钮查看效果，如图 11-5 所示。

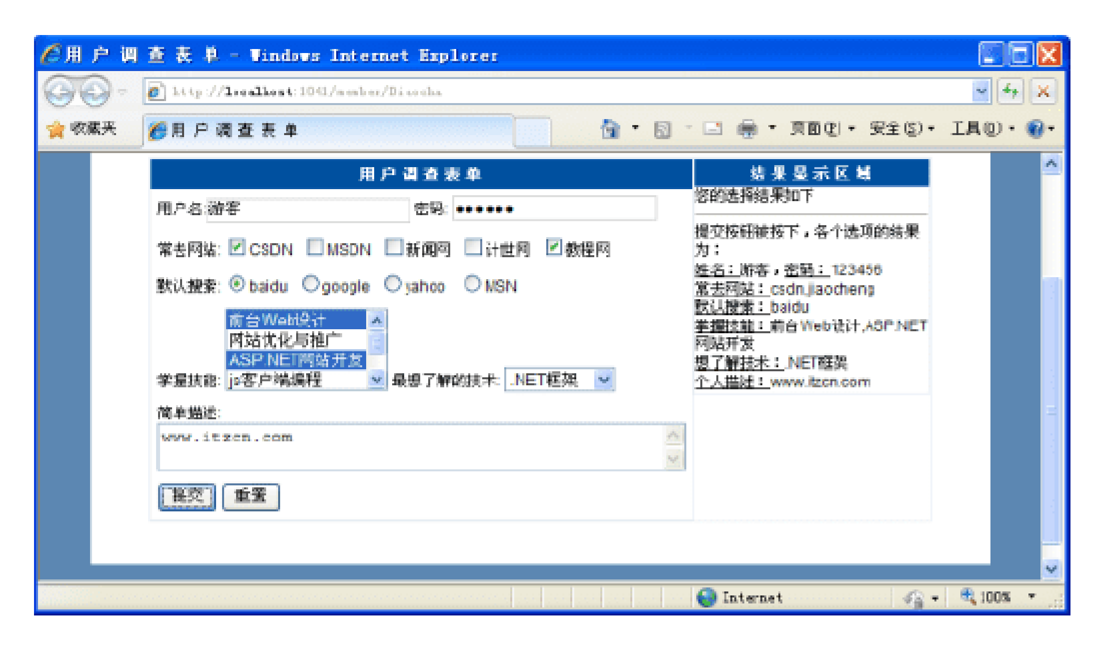


图 11-5　读取与设置表单元素的值

11.4.5　实例分析



源码解析

整个实例主要是使用 val()方法来实现的。具体过程为，先制作表单布局，再使用 val()方法来初始化表单的值，接下来在表单提交时使用 val()方法获取输入的值，然后对这些值进行格式化，最后显示到页面上。

在本实例的各个制作步骤中都给出了详细的解释。但是，大家在使用的时候还应注意区分 val()方法不带参数和带参数的作用。

11.5　可修改字体颜色的新闻查看页

在本节中，我们将看到一个普通的新闻内容展示页面。页面上提供了很多可供选择的颜色块，浏览者可以根据自己的喜好单击它们以改变内容显示的颜色。

视频教学：光盘/videos/11/11.5　可修改字体颜色的新闻查看页　　长度：11 分钟

11.5.1　基础知识——读取/设置 CSS 属性

在实际开发中往往需要对个别的 CSS 属性进行设置。此时，使用 jQuery 提供的 CSS 属性操作方法，可以方便、快捷地读取和设置 CSS 属性。

1. 读取 CSS 属性

在 JavaScript 中，可以使用“对象.style.CSS 属性”的语法来读取或者设置 DOM 元素的 CSS 样式。例如，span.style.color 和 p.style.backgroundColor 等，操作起来非常复杂。

使用 jQuery 中的 css()方法，只需传递一个参数即可获取 CSS 样式。语法格式如下：

```
css(cssName)
```

其中，参数 cssName 表示要获取其值的 CSS 属性名。该方法将从匹配元素集合中获取第 1 个元素的样式属性值并返回。

例如，要获取 p 元素的字体颜色属性，代码如下：

```
$("p").css("color");
```



无论指定的样式属性是通过 style 属性在 HTML 元素中直接设置的，还是通过 CSS 类添加到元素中的，都可以使用 css()方法来读取。

css()方法还考虑到不同浏览器对属性支持的差异，提供了一致的访问接口。例如，对于浮动在右边的元素，下面的代码都将返回字符串 right：

```
$("div").css("float");              //jQuery 方式
$("div").css("cssFloat");           //W3C 标准浏览器
$("div").css("styleFloat");         //IE 浏览器
```

2. 设置 CSS 属性

css()方法的功能非常强大，除了读取 CSS 样式外，还可以设置元素的 CSS 样式。语法格式如下：

```
css(cssName,value)
css(map)
css(cssName,function(index,value))
```

其中，参数 cssName 表示要设置值的 CSS 样式属性名，value 表示要设置的值。如果要设置的值是字符串，则需要用引号括起来；如果值是数字，则不需要引号，而且被默认转换为像素值。

css()方法将对所有匹配元素的样式进行设置。例如，将所有 div 元素的文字设定为白色，并修改背景为黑色，代码如下：

```
$("div").css("color","#FFFFFF");
$("div").css("background-color","balck");
```

也可以一次性设置多个属性的值。下面的代码同时修改所有 div 元素的字体和背景颜色。

```
$("div").css({"color":"#FFFFFF","background-color":"balck"});
```

11.5.2　实例描述

今天本来是周末，本想好好睡一天，谁知老同学小祺打来电话(这段时间让她给弄得都有点不敢接她电话了)。

通过了解才知道她又想加个可修改字体颜色的新闻查看功能(天啊！我只有一天的休息时间啊)。经过软磨硬泡我投降了。

行了，多的咱就不说了，赶紧结束休息日吧。

11.5.3　实例应用

【例 11-5】可修改字体颜色的新闻查看页。

(1) 在 MemberController 中添加名称为 News 的 Action。

(2) 为 News 添加 View，并在返回默认的视图文件中应用母版页。

(3) 引入 jQuery 的类库文件 jquery-1.4.1.min.js。

(4) 制作网页的主体部分，添加一个 ul 列表作为颜色块显示区域。如下所示：

```
<ul>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <li></li>
    <span id="res"></span>
</ul>
```

添加时要注意 HTML 中元素的顺序与结构。其中 ID 为 res 的 span 用来显示用户当前选择的颜色值。

(5) 添加真正显示新闻的布局，有新闻标题、来源、发布时间和具体内容等。

```
<h1>信息时代如何让孩子快乐高效地学习</h1>
<h2>来源：本校宣传部 | 浏览次数：1356 次 | 发布时间：2010-12-8</h2>
```

```
    <span id="text">
        <p>
            信息时代，知识总量在不断增加，孩子学习和接受信息、知识的渠道也越……
        </p>
    </span>
```

在这段代码中需要注意 ID 是 text 的 span 中是新闻内容，它有一个默认颜色，也可以根据用户的选择进行变化。

(6) 在 View 最下方添加 JavaScript 代码，先定义一个颜色数组。

```
<script language="javascript" type="text/javascript">
  //定义要设置的颜色数组
  var colors=new Array("#000","#339","#c93","#6C3","#00F","#0FF","#F66");
</script>
```

(7) 利用 jQuery 强大的选择器将每个颜色应用到所定义的 li 标记中。

```
$(document).ready(function(){
    $("li").each(function(index){                    //遍历 li 元素
        //从颜色数组中设置相应的颜色
        $(this).css("background-color",colors[index]);
    });
});
```

(8) 此时，应该编写代码实现单击过后改变下面的字体颜色。这部分代码如下所示：

```
    $("li").click(function(){
        var
activeCss={"border":1,"border-style":"solid","border-color":"#f00"};
        $(this).css(activeCss);                      //设置当前元素为激活状态
        $("li").not($(this)).css("border","0");      //设置其他元素为不激活状态

    var activeColor=$(this).css("backgroundColor");     //获取当前元素的背景色
        $("text").css("color",activeColor);             //将颜色应用到字体上
        $("#res").html("当前字体的颜色值为："+activeColor)
    });
```

(9) 保存所做的修改。至此，整个实例制作完成。

11.5.4　运行结果

运行项目，在浏览器中输入 member/news 请求查看效果。

待页面打开之后，会看到在一些颜色块下方显示了新闻的内容，如图 11-6 所示。此时，我们可以单击一个颜色块即可更改下面的字体颜色，效果如图 11-7 所示。



图 11-6　默认运行效果

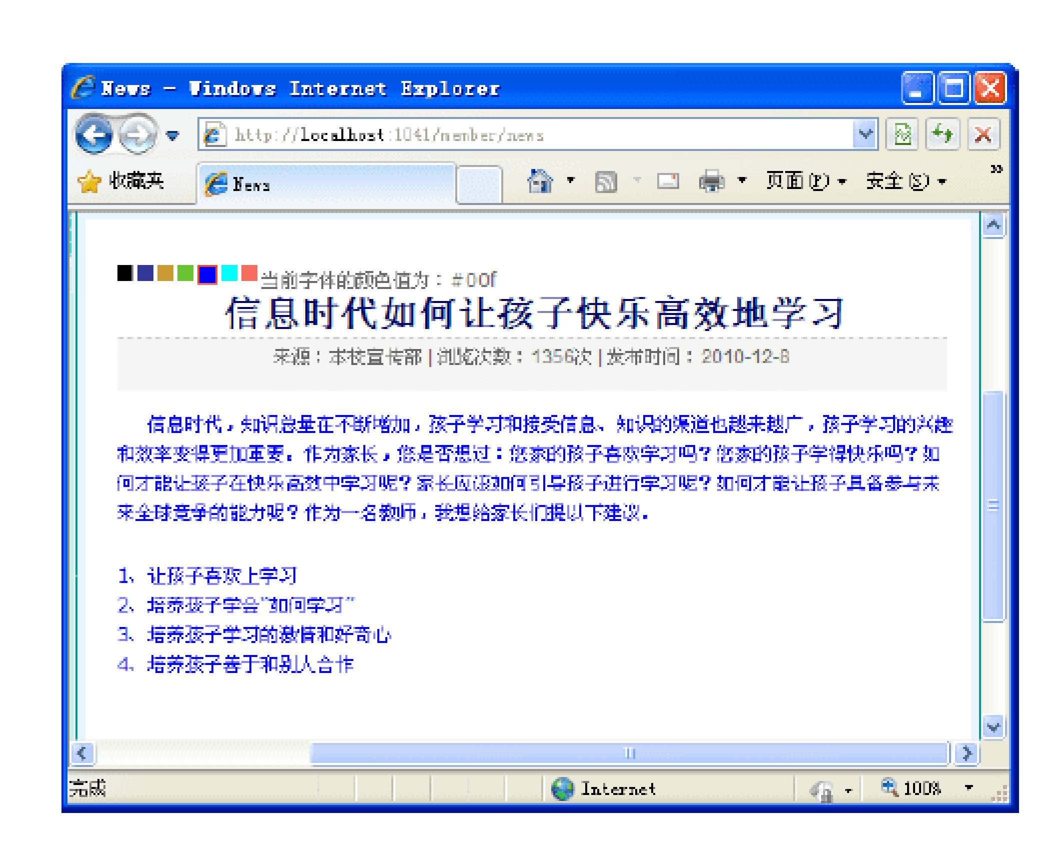


图 11-7　改变颜色后运行效果

11.5.5　实例分析



源码解析

本实例先定义了一个颜色数组 colors，然后使用 css()方法依次将每个颜色值应用到页面的 li 标记上形成颜色块。这一步应用的 CSS 样式属性是 background-color。

单击颜色块时，应用 css()方法修改了当前 li 标记的 border、border-style 和 border-color 属性，使其突出显示。之后，使用 css()方法清除其他颜色块的 border 属性。

剩下的操作就是获取当前的颜色值，并将它应用到下面的文字上。

11.6　横向滑动的下拉菜单

任何由多个页面组成的网站都需要某种导航系统。所谓导航系统其实就是让浏览者能够知道自己在当前网站里所处的位置，并快速返回原来的位置或者快速找到你想要进入的页面。

制作网站导航系统的方法有很多种，其中使用下拉菜单来组织站点结构是最常见的办法。下面来看看如何用 jQuery 实现这种结构，即添加滑动菜单效果。



视频教学：光盘/videos/11/11.6　横向滑动的下拉菜单　　　　长度：11 分钟

11.6.1　基础知识——jQuery 动画效果

动画效果也是吸引众多开发人员眼球的一道风景。通过 jQuery 的动画方法，能够轻松地为网页添加非常精彩的视觉效果，给用户一种全新的体验。例如，隐藏或显示网页的菜单效果，网页部分元素的渐变效果，滑动效果等。

1. 显示与隐藏效果

在网页上实现一个元素的显示或者隐藏效果，是项目开发中经常遇到的问题。在 jQuery

中可以十分容易地实现这种效果，主要使用了 hide()、show()和 toggle()方法。

表 11-11 中列出了这 3 种方法的语法说明及示例。

表 11-11　显示与隐藏效果

名　称	语法说明	示　例
hide()	隐藏匹配元素	$("#txt").hide();
hide(speed,[callback])	以指定的 speed 隐藏所有匹配的元素，并在完成后可选地触发一个回调函数 callback	$("#img").hide(7000);
show()	显示被隐藏的匹配元素	$("#txt").show();
show(speed,[callback])	以指定的 speed 显示所有匹配的元素，并在完成后可选地触发一个回调函数 callback	$("#img").show(7000);
toggle()	如果元素是可见的，切换为隐藏的；如果元素是隐藏的，切换为可见的	$("#switch").toggle();
toggle(switch)	根据 switch 参数切换元素的可见状态(true 为可见，false 为隐藏)	$("li").toggle(id++ % 2 == 0);

2. 滑动效果

jQuery 可以实现的滑动效果包括向上收缩、向下展开和交替伸缩样式(上下滑动)。

1) 向上收缩

通过高度变化(向上减小)来动态地隐藏所有匹配的元素，隐藏完成后可选地触发一个回调函数。

```
slideUp(speed,[callback])
```

这个动画效果只调整元素的高度，可以使匹配的元素以滑动的方式隐藏起来。其中，参数 speed 表示 3 种预定速度之一的字符串(slow、normal 及 fast)或表示动画时长的毫秒数值(如 1000)；参数 callback 是可选的，表示在动画完成时执行的函数。

2) 向下展开

通过高度变化(向下增大)来动态地显示所有匹配的元素，显示完成后可选地触发一个回调函数。

```
slideDown(speed,[callback])
```

这个动画效果只调整元素的高度，可以使匹配的元素以滑动的方式显示出来。其中，参数 speed 表示 3 种预定速度之一的字符串(slow、normal 及 fast)或表示动画时长的毫秒数值(如 1000)；参数 callback 是可选的，表示在动画完成时执行的函数。

3) 交替伸缩

通过高度变化来切换所有匹配元素的可见性，并在切换完成后可选地触发一个回调函数。

```
slideToggle(speed,[callback])
```

这个动画效果只调整元素的高度，可以使匹配的元素以滑动的方式隐藏或显示。其中，参数 speed 表示 3 种预定速度之一的字符串(slow、normal 及 fast)或表示动画时长的毫秒数值(如 1000)；参数 callback 是可选的，表示在动画完成时执行的函数。

3. 淡入淡出效果

淡入淡出效果也是很多网站上面都有的图片动画效果之一，jQuery 对此也提供了支持，还可以自定义图片的不透明度。

1) 淡入淡出效果

通过不透明度的变化来实现所有匹配元素的淡入效果，并在动画完成后可选地触发一个回调函数。这个动画只调整元素的不透明度，也就是说所有匹配的元素的高度和宽度不会发生变化。

```
fadeIn(speed,[callback])
```

其中，参数 speed 表示 3 种预定速度之一的字符串(slow、normal 及 fast)或表示动画时长的毫秒数值(如 1000)；参数 callback 是可选的，表示在动画完成时执行的函数。

2) 淡出效果

通过不透明度的变化来实现所有匹配元素的淡出效果，并在动画完成后可选地触发一个回调函数。这个动画只调整元素的不透明度，也就是说所有匹配的元素的高度和宽度不会发生变化。

```
fadeOut(speed,[callback])
```

其中，参数 speed 表示 3 种预定速度之一的字符串(slow、normal 及 fast)或表示动画时长的毫秒数值(如 1000)；参数 callback 是可选的，表示在动画完成时执行的函数。

3) 自定义不透明度

把所有匹配元素的不透明度以渐进方式调整到指定的不透明度，并在动画完成后可选地触发一个回调函数。这个动画只调整元素的不透明度，也就是说所有匹配的元素的高度和宽度不会发生变化。

```
fadeTo(speed,opacity,[callback])
```

其中，参数 speed 表示三种预定速度之一的字符串(slow、normal 及 fast)或表示动画时长的毫秒数值(如 1000)；参数 opacity 表示要调整到的不透明度值(0～1 之间的数字)；参数 callback 是可选的，表示在动画完成时执行的函数。

11.6.2 实例描述

一直感觉 jQuery 的动画效果很神秘，有一种望而却步的感觉。今天，我终于冲破自己的极限，去挑战它了。

结果是，它没有想象得那么难，下面把自己的战绩与大家分享一下！

11.6.3 实例应用

【例 11-6】横向滑动下拉菜单。

(1) 在 MemberController 中添加名称为 Menu 的 Action。

(2) 为 Menu 添加 View，并在返回默认的视图文件中应用母版页。

(3) 引入 jQuery 的类库文件 jquery-1.4.1.min.js。

(4) 制作网页的主体部分，即横向下拉菜单的布局结构。整体代码如下：

```
    <ul>
        <li class="hmain"><a href="#">教育科研</a>
            <ul>
                <li><a href="#">教研动态</a> </li>
                <li><a href="#">教师论文</a> </li>
                <li><a href="#">课件交流</a> </li>
                <li><a href="#">外出学习体会</a> </li>
            </ul>
        </li>
        <!-- 此处省略其他菜单的布局结构 -->
    </ul>
```

(5) 编写一些 CSS 样式来修饰上面的菜单，包括取消默认的项目符号，为主菜单指定一个背景图片，设置菜单文字的字体以及显示的宽度等。

```
        ul, li
        {
            /*清除 ul 和 li 上默认的小圆点*/
            list-style: none;
        }
        ul
        {
            /*清除子菜单的缩进值*/
            padding: 0;
            margin: 0;
        }
        .hmain
        {
            background-image: url(/images/title.gif);
            background-repeat: repeat-x;
            width: 120px;
        }
        a
        {
            /*取消所有的下划线*/
            text-decoration: none;
            padding-left: 20px;
            display: block;
            display: inline-block;
            width: 100px;
            padding-top: 3px;
            padding-bottom: 3px;
        }
        li
        {
            background-color: #EEEEEE;
        }
        .hmain a  /*主菜单标题的样式*/
        {
            color: white;
            background-image: url(/images/collapsed.gif);
            background-repeat: no-repeat;
            background-position: 3px center;
        }
        .hmain li a/*子菜单链接的样式*/
```

```
    {
       color: black;
       background-image: none;
    }
    .hmain ul
    {
       display: none;
    }
    .hmain
    {
       float: left;
       margin-right: 1px;
    }
```

(6) 根据前面定义的菜单结构，利用 jQuery 强大的选择器功能以及内置的动画方法，可以很容易地实现将鼠标移到菜单标题上时向下展开子菜单，移出时向上收缩子菜单的效果。具体实现代码如下：

```
<script language="javascript" type="text/javascript">
    $(document).ready(function () {
        $(".hmain").hover(function () {                    //鼠标移到上面时触发
            //将当前元素下的 ul 列表向下展开
            $(this).children("ul").slideDown();
            changeIcon($(this).children("a"));
        }, function () {                                   //鼠标移出时触发
        //将当前元素下的 ul 列表向上收缩
            $(this).children("ul").slideUp();
            changeIcon($(this).children("a"));
        });
    })
</script>
```

(7) 上述代码调用了 changeIcon()函数，该函数实际上是一个辅助函数。用于根据展开或者收缩状态动态地修改主菜单的指示图标。

```
// 修改主菜单的指示图标
function changeIcon(mainNode) {
    if (mainNode) {
        if (mainNode.css("background-image").indexOf("collapsed.gif") >= 0)
{
            mainNode.css("background-image",
"url('/images/expanded.gif')");
        } else {
            mainNode.css("background-image",
"url('/images/collapsed.gif')");
        }
    }
}
```

(8) 保存所做的修改。至此，整个实例制作完成。

11.6.4 运行结果

运行项目，在浏览器中输入 member/menu 请求查看效果。

待页面打开之后，会看到默认运行后所有的子菜单都不显示，处于折叠状态，如图 11-8 所示。我们可以将鼠标移到要查看的菜单标题上，其包含的子菜单将以向下展开动画慢慢出现。

图 11-9 为从“教工之家”菜单移动到“学生园地”菜单时的动画效果。“教工之家”菜单正在向上收缩，而“学生园地”菜单则正在向下展开。



图 11-8　默认运行后全部折叠



图 11-9　展开“学生园地”菜单时动画效果

11.6.5　实例分析



源码解析

本实例使用了嵌套的 ul 列表结构来定义主菜单和子菜单。

最外层的 ul 表示整个菜单区域，内部每个 class 为 hmain 的 li 表示一个下拉菜单。在 a 标记中是主菜单的标题，紧接下来的 ul 列表中是其子菜单。在这里仅给出了一个菜单的结构，大家可以根据格式进行自由扩展。

在实现动画效果时，以主菜单的标题为目标，当鼠标移上或者移出时分别调用 slideDown()和 slideUp()方法，显示动画效果。

当然，在这里读者也可以添加其他的动画效果。例如慢慢淡入出现，淡出消失等。

11.7　定制一个中文日历

在 Web 页面中实现一些常用的页面特效常常会让 Web 程序开发人员苦恼不已。

jQuery UI 将一些常用的页面组件封装成简单的类库供程序开发人员使用，包括折叠面板、日期选择器和对话框组件等。

这些组件不仅具有多种配置选项，而且是主题化的。使用它们可以让程序开发人员编写简单的代码就能实现丰富的页面效果。

下面我们将使用 jQuery UI 中的日期选择器组件来实现一个中文日历。



视频教学：光盘/videos/11/11.7　定制一个中文日历　　长度：11 分钟

11.7.1　基础知识——UI 库日期选择器组件

jQuery UI 中的日期选择器组件用于在页面输入框上添加选择日期的功能，以便用户输入日期类型的数据。可以自定义日期格式、语言以及限制可选日期的范围，还可以轻松地添加按钮和其他导航选项。

要使用日期选择器组件，只需在使用 jQuery 选择器匹配一个页面元素，然后调用该对象扩展的 datepicker()方法即可。该方法的格式如下：

```
$(selector).datepicker([options]);
```

选项 selector 是选择对象的选择器，参数 options 是初始化日期选择器组件的配置选项集合。表 11-12 中给出了最常用的配置选项及说明。

表 11-12　常用配置选项

名　称	说　明
dateFormat	指定解析和显示日期的格式，默认值是 mm/dd/yy(月/日/年)
dayNames	设置长日期名字列表。从星期日开始，按照 dateFormat 的设置来使用
dayNamesMin	设置短日期名字列表。从星期日开始，用在日期选择器每列的标题部分
firstDay	设置一周的第一天，星期天是 0，星期一是 1
maxDate	可选的最大日期
minDate	可选的最小日期
monthNames	完整的月份名称列表。用在日期选择器的月份列表的标题部分
showMonthAfterYear	指定是否在标题的年份后面显示月份
showWeek	设置是否显示一年中的周数
yearSuffix	指定月份标题中年份之后的附加文本

11.7.2　实例描述

默认情况下，日历是按英语语种国家的操作习惯布局的，如图 11-10 所示。



图 11-10　默认的日期选择器界面

中国人的传统习惯与其有很大区别。例如，我国一直以星期一为每星期的第一天，日期总是按从大到小的顺序排列(比如：××××年××月××日)，而且界面也以中文和阿拉伯数字为主。

下面这个实例将对默认的日历进行本地化配置，打造一个符合中文习惯的日历。

11.7.3　实例应用

【例 11-7】定制一个中文日历。

(1) 在 MemberController 中添加名称为 Blog 的 Action。

(2) 为 Blog 添加 View，并在返回默认的视图文件中应用母版页。

(3) 引入 jQuery 的类库文件 jquery-1.4.1.min.js。

(4) 由于日历组件属于 jQuery 的第三方组件被放在 jQuery UI 库中，所以这里需要先下载该库文件，并放在与 jQuery 类库相同的位置。

这里以 jQuery UI 1.8.5 为例，引入代码如下：

```
    <script src="../../Scripts/jquery-ui-1.8.5.min.js"
type="text/javascript"></script>
```

(5) 同时还需要引用在 jQuery UI 库中提供的 CSS 样式文件，代码如下：

```
    <link href="../../jQueryUI/jquery.ui.all.css" rel="stylesheet"
type="text/css" />
```

(6) 在页面的合适位置添加如下代码，以显示一个日期输入文本框。

```
    <h2>查看 2010 年教学计划安排</h2>
    请输入一个日期：<input type="text" id="datepicker" />
```

(7) 这一步编写 jQuery 代码，在单击日期文本框时显示一个日历，并通过 datepicker 提供的选项配置成中文。

```
    <script type="text/javascript" language="javascript">
        $(document).ready(function () {
            $("#datepicker").datepicker({
                /* 区域化周名为中文 */
                dayNamesMin: ["日", "一", "二", "三", "四", "五", "六"],
                /* 每周从周一开始 */
                firstDay: 1,
                /* 区域化月名为中文习惯 */
                monthNames: ["1 月", "2 月", "3 月", "4 月", "5 月", "6 月","7 月", "8
月", "9 月", "10 月", "11 月", "12 月"],
                /* 月份显示在年后面 */
                showMonthAfterYear: true,
                /* 年份后缀字符 */
                yearSuffix: "年",
                /* 格式化中文日期
                (因为月份中已经包含“月”字，所以这里省略) */
                dateFormat: "yy 年 MMdd 日"
            });
        });
    </script>
```

(8) 保存所做的修改。至此，整个实例制作完成。

11.7.4　运行结果

运行项目，在浏览器中输入 member/blog 请求查看效果。单击使文本框获得焦点，此时将在下方弹出一个日历面板，从中选择一个日期后将显示在文本框内，如图 11-11 所示。



图 11-11　中文日历选择与运行效果

11.7.5　实例分析



源码解析

从本实例可以看到，jQuery UI 库继承了 jQuery 简单的特点。

要显示一个日历，只需在选中元素后调用 datepicker()方法即可。为了使其显示符合中文习惯，该方法通过各种参数进行了设置。

包括设置在日历面板中显示周名、每周的第一天，在日期选择器面板显示的月份名称、月份和年份的位置习惯，在日期选择器面板中的年份的后缀，以及输出的日期格式等。

11.8　浮动的注册条款

在进行一些逻辑处理的时候，难免会用到对话框，就像 QQ 空间和一些博客、论坛都开始大量使用页面浮动层实现的模拟对话框(见图 11-12)。这种方式非常自由，开发者可以自主控制对话框的布局、内容和样式，弹出对话框也非常时尚、美观。

假如现在某个网站还在使用最古董的 JavaScript 函数弹出对话框(见图 11-13)，那就只能说明这个站点的技术人员已经跟不上时代的步伐了。

当然，开发人员完全可以自己编写 JavaScript 代码和 CSS 样式表来实现一个自定义的对话框。但是，真正着手编写过自定义对话框的人可能会深有体会：这活儿确实费工夫！

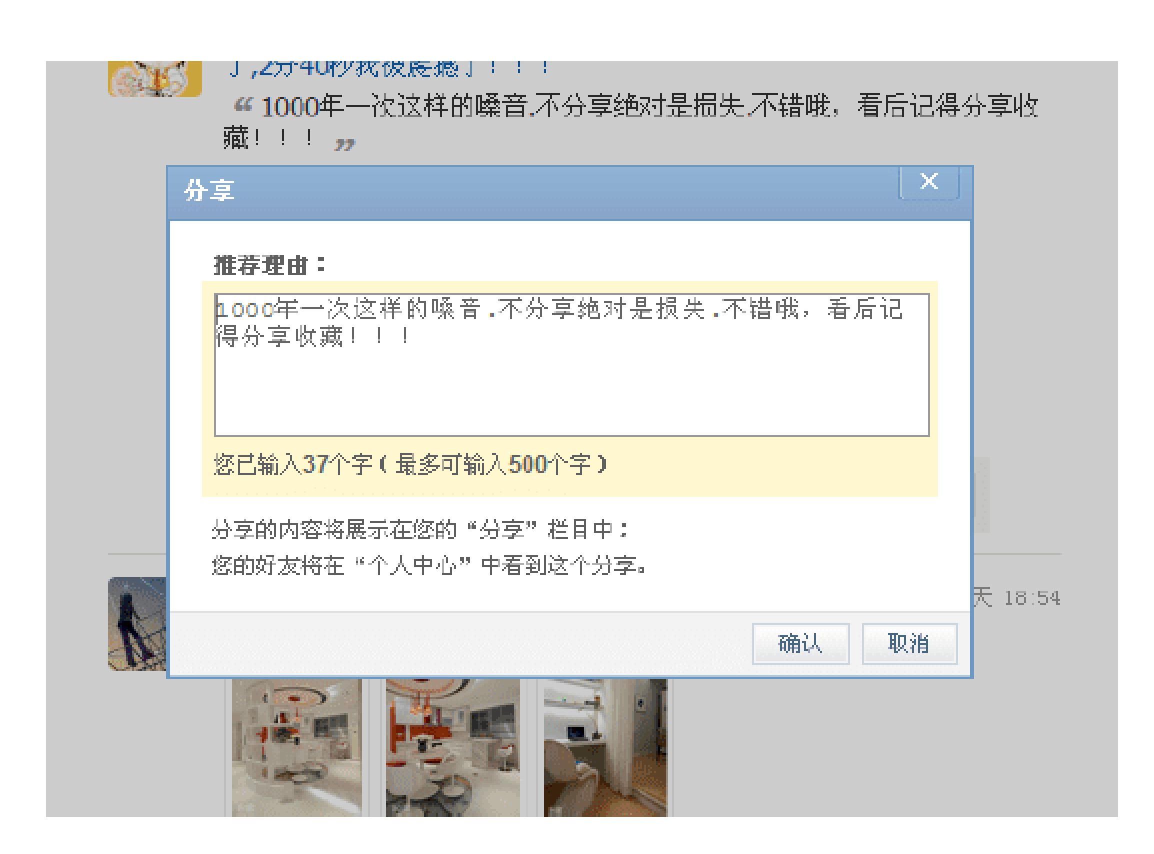


图 11-12　QQ 空间自定义的【分享】对话框

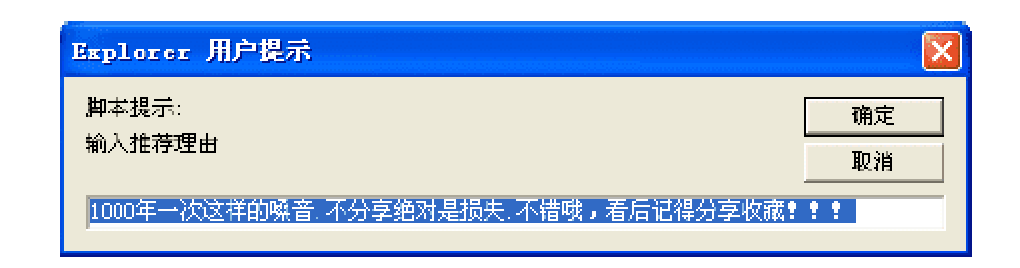


图 11-13　JavaScript 函数弹出的对话框

值得庆幸的是，jQuery UI 提供了一个现成的对话框组件，非常易用、功能强大，而且非常美观、绚丽多彩。

视频教学：光盘/videos/11/11.8　浮动的注册条款　　　长度：10 分钟

11.8.1　基础知识——UI 库对话框组件

jQuery UI 对话框组件可以显示文本、图片和多媒体，甚至还有交互式表单，而且该对话框组件还可以随意地在页面内拖动和调整大小。默认情况下，可以通过对话框右上角的☒按钮关闭该对话框。

使用 jQuery 选择器匹配一个页面元素，然后调用该对象扩展的 dialog()方法即可。该方法格式如下：

```
$(selector).dialog([options]);
```

参数 options 是初始化对话框组件的配置选项集合。配置选项的具体说明如表 11-13 所示。

表 11-13　对话框组件的配置选项

名　称	说　明
autoOpen	是否自动打开对话框
closeOnEscape	指定在获得焦点，并且当用户按下 Esc 键的时候是否关闭该对话框
closeText	指定关闭按钮的文本
draggable	是否通过拖动标题栏来移动对话框
height	设置对话框的高度，以像素为单位。设置为 auto 则说明让对话框根据内容自动调整高度
maxHeight	设置对话框可以调整到的最大高度。默认值为 false，表示不限制
maxWidth	设置对话框可以调整到的最大宽度。默认值为 false，表示不限制
minHeight	设置对话框可以调整到的最大高度。默认值为 150
minWidth	设置对话框可以调整到的最大宽度。默认值为 150
modal	是否为模式对话框

续表

名　称	说　明
position	设置对话框的初始位置
resizable	是否可以调整大小。如果为 true，对话框可以调整大小。默认值为 true
title	指定对话框的标题
sidth	设置对话框的宽度，以像素为单位。默认值为 300

11.8.2　实例描述

今天在 Web 开发技术群中，有个昵称为“.NET 小白菜”的网友(我的徒弟)问了这样一个问题：如何实现网页上流行的浮动对话框，类似 QQ 空间那种？

我给他的解决方案是利用 jQuery UI 库来实现，这里给大家讲一下。如果有写得不好的地方，欢迎大家提建议。

11.8.3　实例应用

【例 11-8】浮动的注册条款。

(1) 在 MemberController 中添加名称为 Dialog 的 Action。

(2) 为 Dialog 添加 View，并在返回默认的视图文件中应用母版页。

(3) 引入 jQuery 的类库文件 jquery-1.4.1.min.js。

(4) 引入 jQuery UI 的类库和 CSS 文件。

```
    <script src="../../Scripts/jquery-ui-1.8.5.min.js"
type="text/javascript"></script>
    <link href="../../jQueryUI/jquery.ui.all.css" rel="stylesheet"
type="text/css" />
```

(5) 制作网页的主体部分，添加网站服务条款，代码如下：

```
    <div class="content">
        <h1>网站服务条款</h1>
        <span id="text">
            <p>
                欢迎您加入本站点参加交流和讨论，为维护网上公共秩序和社会稳定，请您自觉遵守
以下条款： <br />   <br />
                1.服务条款的确认和接纳<br />
                2.服务简介<br />
                3.服务条款的修改<br />
                4.服务修订<br />
                5.用户隐私制度<br /> <br />
                <input id="Button1" type="button" value="同意(关闭)" />
            </p>
        </span>
    </div>
```

(6) 编写 jQuery 代码使上一步添加的布局代码在页面打开后，以模态对话框的浮动形式显示。代码如下：

```
<script language="javascript" type="text/javascript">
    $(document).ready(function () {
        $(".content").dialog({
            modal: true,        //启用模态对话框功能
            width: 600          //定义对话框的宽度
        });
    });
</script>
```

(7) 保存所做的修改。至此，整个实例制作完成。

11.8.4　运行结果

运行项目，在浏览器中输入 member/dialog 请求查看效果。待页面打开之后，会以浮动的形式显示一个对话框，其中包含了在页面中定义的网站服务条款。

该对话框可以自由拖动，默认带有一个关闭按钮，效果如图 11-14 所示。

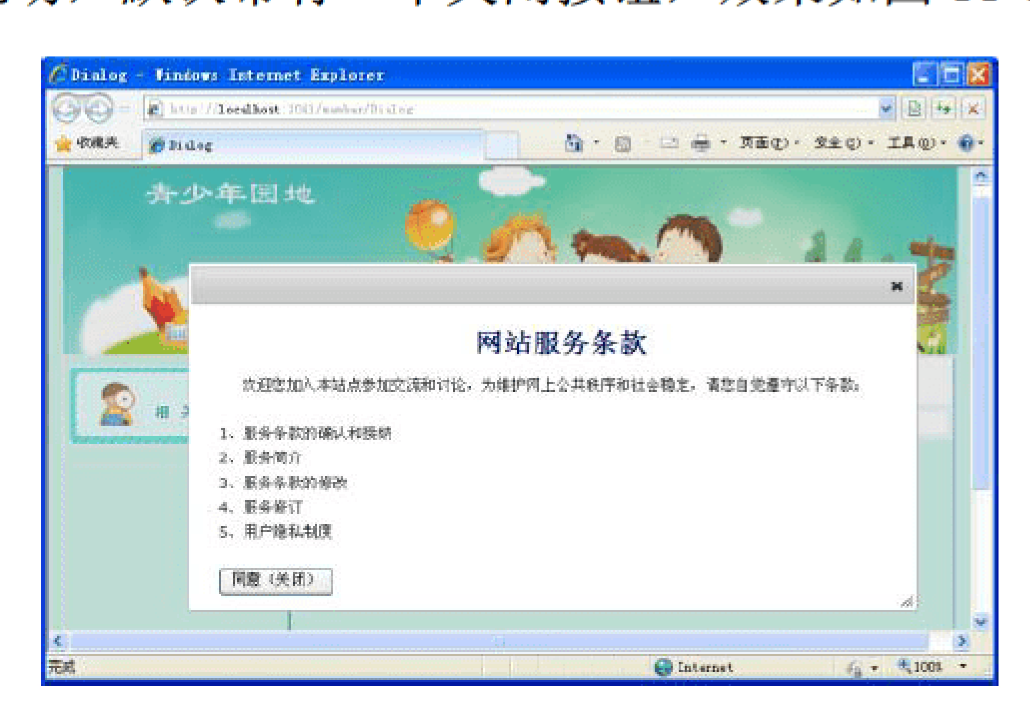


图 11-14　浮动注册条款运行效果

11.8.5　实例分析



源码解析

本实例主要使用了 dialog 组件的 model 属性来显示一个模态对话框。这在需要中断用户操作，等待用户处理完该次中断以后才能继续对页面上的内容进行操作时非常有效。

dialog 组件还提供了很多方法与事件来处理用户在单击或者关闭对话框时触发的操作。

11.9　常见问题解答

11.9.1　如何给列表的偶数行添加背景色



如何给列表的偶数行添加背景色?

网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

现在有一个列表，我想给它的偶数行加上背景色，怎么用 jQuery 实现它呢？

【解决办法】参考以下列表。

```
<ul>
  <li>新闻</li>  <li>网页</li>
  <li>视频</li>  <li>空间</li>
  <li>图片</li>  <li>地图</li>
  <li>更多</li>
</ul>
```

用 jQuery 去实现给列表的偶数行添加背景色，代码如下：

```
  $(document).ready(function(){
    $("ul li:even").css("background","#FCF");
     })
```

这样问题就解决了。

11.9.2 怎样得到 jQuery 数组对象中的某个对象



怎样得到 jQuery 数组对象中的某个对象？

网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

怎么得到 jQuery 数组对象中的某个对象？例如，在一个文档中我放了 5 张图片，我的 jQuery 代码如下：

```
$(document).ready(function ()
  {
  var allimg=$("img");
  }
);
```

现在我想得到第 4 个 img，并想获得它的 src 属性。那么我应该怎么做？我用 allimg[4]来获得结果失败，用下面代码：

```
var allimg=new Array();
allimg=$("img");
```

结果还是不行。又用 window.alert(allimg[4].attr("src"));来做一个例子，结果不成功。请不要告诉我用 ID 选择器来做啊……

我只是想知道一种方法，如果用 ID 的话那么就要对每个 Img 设置。

【解决办法】其实楼主离成功已经很近了。正确的语句为：

```
  alert($('img').eq(3).attr('src'));
```

在这个例子中，allimg 返回的是 jQuery 对象数组。allimg[3]返回的是 html 的一个对象，html 对象没有 attr()方法。allimg.eq(3)返回的是 jquery 的一个对象，并且具有 attr()方法，所以它才是正解的。

是不是有些眉目了？你好好琢磨吧。

11.9.3　怎样用 jQuery 获取具有相同 class 的 text 值



怎样用 jQuery 获取具有相同 class 的 text 的值呢？
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

在 jQuery 中怎样获取具有相同 class 的 text 的值呢？

【解决办法】如果你会使用单个的 text，那么这个问题应该难不住你。看看答案吧：

```
$(".className").each(function(){
    alert($(this).text());
});
```

11.9.4　如何让 jQuery 图片延长 2 秒显示



如何让 jQuery 图片延长 2 秒显示
网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

我现在想在页面加载后，将图片延长 2 秒显示出来，怎么写？

【解决办法】HTML 代码如下：

```
<center>
  <img id="myimg" src="http://www.baidu.com/img/baidu logo.gif" />
</center>
```

jQuery 代码如下：

```
$(document).ready(function(){
 $('#myimg').css('display', 'none');
 $('#myimg').load(function(){
  alert('图片加载完毕，在单击确认 2 秒后，图片出现。');
  $(this).delay(2000).fadeIn(400);
 });
});
```

这样就解决了。

11.10　习　　题

一、填空题

(1) jQuery 的基本选择器包含 CSS 选择器、__________和表单域选择器。

(2) 可以在 CSS 选择器中使用__________来选择 HTML 页面中已有的标签元素。

(3) 假设要查找 menu 元素下所有 item 的子元素，应该使用__________代码。

(4) 在表单域选择器中，用__________选择所有不可见元素以及隐藏域。

(5) __________选择器根据索引值对元素进行筛选，且以冒号:开头。

(6) 如果 are 元素及其父元素在文档中不占用空间，则可以用__________选择器来过滤。

(7) 假设有如下 HTML 代码:

```
<ul>
<li>1</li> <li>2</li> <li>3</li> <li>4</li> <li>5</li>
</ul>
```

现在要筛选出第 4 个元素，并将它的背景色设置为#FCF，应该使用代码__________。

(8) 将所有 div 元素的文字设定为白色，并修改背景为黑色，代码为__________。

二、选择题

(1) 假设有如下一行代码:

```
<input type="checkbox" name="city"  value="zhengzhou">
```

现在要获取其中的值，应该使用代码__________。

A. $("input:checkbox[name='city']").val("anyang")

B. $("input:checkbox[name='city']").val()

C. $("input:checkbox:checked").val()

D. $("input ").val()

(2) 下列选项中不属于 CSS 选择器的是__________。

A. $("p").css("color","#F00")

B. $(".title").css("color","#F00")

C. $("#title").css("color","#F00")

D. $("#title >p").css("color","#F00")

(3) 要搜索跟在每个匹配元素之后的所有同辈元素，应该选择如下的__________搜索方法。

A. nextAll([selector])　　B. nextUntil([selector])

C. siblings([selector])　　D. next([selector])

(4) 假设在表单中有一个城市列表 city，希望选中“郑州”和“安阳”两个，应该使用下列哪一句代码? __________

A. $("city").val(["郑州","安阳"]);

B. $("city").val("郑州","安阳");

C. $("city").value("郑州","安阳");

D. $("city").value({"郑州","安阳"});

(5) 使用 jQuery 在选中一个元素后想实现其显示与隐藏效果，应该使用__________方法。

A. hide()　　B. show()

C. toggle()　　D. slideToggle()

(6) 下面哪一个值不可以作为 fadeOut()方法的值? __________

A. disable　　B. slow

C. normal　　D. 1000

(7) 在使用日历组件 datepicker 时，利用__________属性可以设置长日期名字列表。

A. dateFormat　　B. dayNamesMin

C. showMonthAfterYear　　D. dayNames

三、上机练习

上机练习 1: 修饰表单。

在本章的第 11.1 节中我们学习了各种 jQuery 选择器的基本语法及简单示例。这个上机练习的重点也是选择器，要求读者创建一个普通的表单，然后利用选择器找到表单元素并使用 CSS 样式进行修饰。

图 11-15 为最终运行效果，其中包含了文本输入框、按钮、复选框和列表框等。

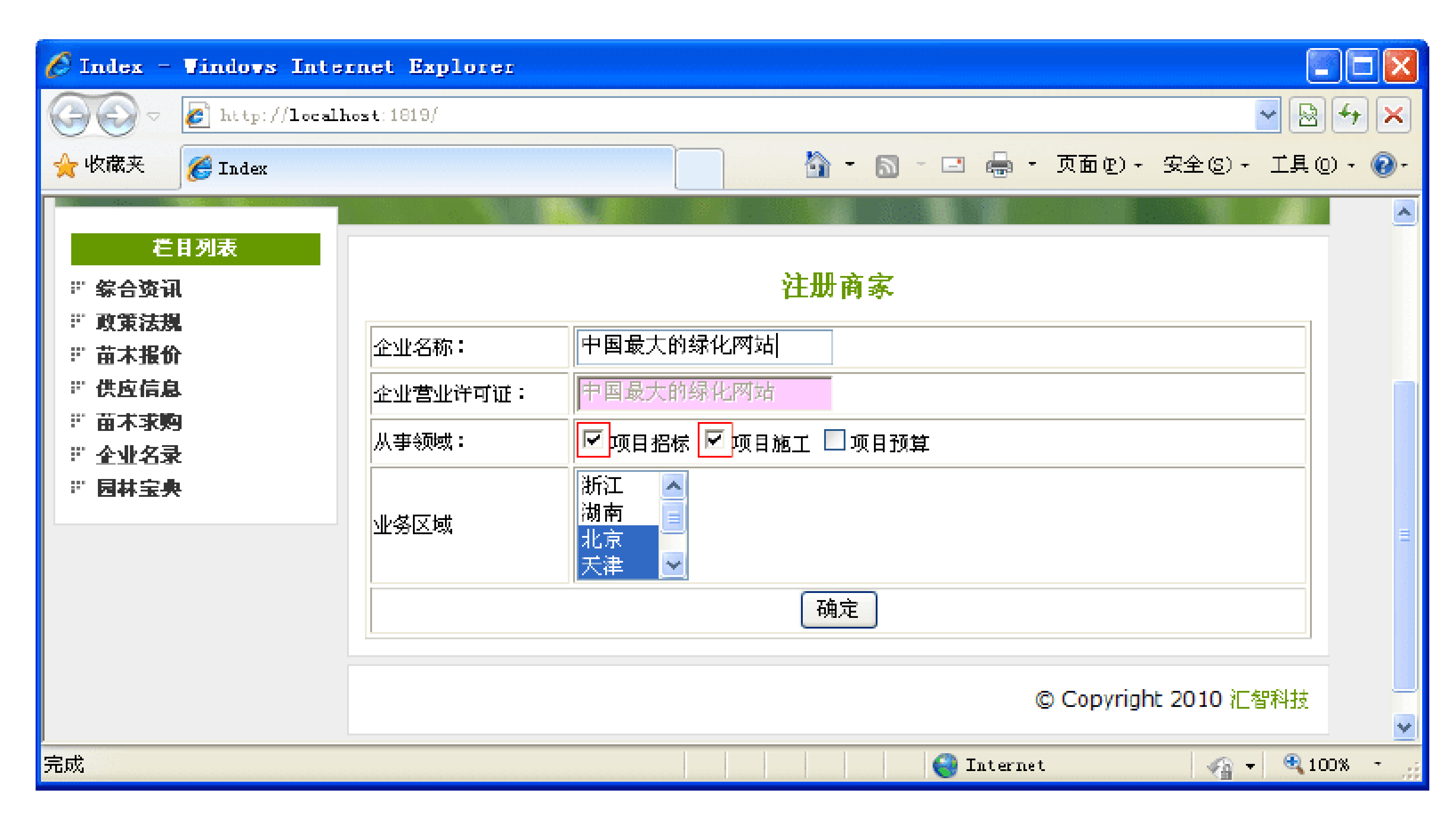


图 11-15　运行效果

上机练习 2: 实现纵向下拉菜单。

本次练习要求读者实现纵向下拉菜单的滑动效果，如图 11-16 所示。步骤可以为：先创建下拉菜单的结构，再用 CSS 实现纵向排列，然后使用 jQuery 对主菜单标题进行修改，接下来隐藏子菜单，最后在单击的时候控制其隐藏与显示。



图 11-16　运行效果



第 12 章　注入 Ajax 特性的 MVC

内容摘要

Ajax 即 Asynchronous JavaScript and XML，中文意思是“异步 JavaScript 和 XML”。它所用到的技术早已有之，近年来随着 Web 2.0 的发展，它也跟着风靡开来。有人说它新瓶装旧酒，这里我们不管它是新酒还是旧酒，喝着过瘾就行。

如果网站上没有实现 Ajax，那么证明这个网站还不够酷。至少，这是很多人对 Web 2.0 之后的开发世界的主要印象。对于大部分情况，实现网站上的 Ajax 功能有很多方法。大部分必需的技术封装在所选择的脚本库中，但是不一定要求如何构建服务器代码以确保它遵循了 MVC 且易于修改。

对于下面即将展示的示例，将遵循相当一致的模式来构建服务器端的代码。

- 每个 Ajax 例程将在控制器上使用一个专用动作(Action)。
- 动作将检查入站请求是不是一个 Ajax 请求。
- 除非动作不是一个 Ajax 请求，否则每一个动作都将返回一个专用的视图。

Ajax 的优点就是不用刷新页面，使用户体验更具连贯性。对于单击操作，在系统返回结果之前，页面没有任何变化。该看电影看电影，该读小说读小说，无聊了看看广告也比盯着白屏好。接到系统返回结果了，页面该变的地方变一下，换成我们想要的东西。我们还是该干吗干吗，基本上完全避免了那个傻乎乎等待的过程。

本章将讲解 ASP.NET MVC 中的 Ajax。

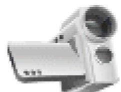
学习目标

- 了解并掌握 XMLHttpRequest 对象
- 了解并掌握$.get()方法
- 了解并掌握$.post()方法
- 了解并掌握$.ajax()方法
- 了解并掌握如何异步请求 JSON 数据
- 了解并掌握 Ajax.BeginForm()
- 了解并掌握 Ajax 全局事件

12.1 异步访问控制器动作

在 ASP.NET MVC 中，每个 Request 都被 route 到一个 Controller 下的 Action 来处理，即一个 Controller Class 的方法。因此，如果在 Action 方法中完成业务逻辑，并把需要回传的数据写回到 Response 中，在客户端再由 JavaScript 来处理这些回传的数据，相信也能实现 Ajax。

jQuery 对 JavaScript 中的 Ajax 进行了封装。jQuery 中的 Ajax 实现方法最大的优势就是消除了各个浏览器之间的差异，提高了程序的兼容性。其次它简化了开发人员的操作，减少了程序代码量。

**视频教学**：光盘/videos/12/12.1 异步访问控制器动作　　长度：14 分钟

12.1.1 基础知识——XMLHttpRequest 对象

XMLHttpRequest 对象是 Ajax 的核心，用于实现客户端脚本与服务器之间的数据交互过程。它并非最近才出现，微软最先在 Internet Explorer 5.0 中作为一个 ActiveX 对象引入了 XMLHttpRequest 对象。目前主流的浏览器都已经对 XMLHttpRequest 对象提供了良好的支持，像 FireFox、Safari、Chome 和 Opera 等。

例如，如下给出的代码会创建一个跨浏览器的 XMLHttpRequest 对象。在执行创建 XMLHttpRequest 对象的时候，会对浏览器的支持进行判断，获取相应的 XMLHttpRequest 对象。具体创建代码如下：

```
var xmlRequest;
if(window.ActiveXObject)
{
    /* 如果浏览器支持 ActiveX 对象,
        就使用 ActiveX 对象创建一个 XMLHttpRequest 对象
        支持 IE 浏览器
    */
    xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
}
else if(window.XMLHttpRequest)
{
    /* 如果浏览器支持 XMLHttpRequest 对象
        就直接创建一个 XMLHttpRequest 对象, 以
        支持其他主流浏览器(FF,Chome,Safari 等)
    */
    xmlRequest = new XMLHttpRequest();
}
```

无论采用何种方式创建，XMLHttpRequest 对象的使用方法都是相同的。XMLHttpRequest 对象在浏览器中通过 HTTP 协议与服务器进行数据交换，像字符串数据和 XML 数据等。

XMLHttpRequest 对象提供了很多属性和方法来处理和控制 HTTP 请求与响应。表 12-1 中列出了 XMLHttpRequest 对象的属性以及简要说明。

表 12-1　XMLHttpRequest 对象的属性

名　称	说　明
onreadystatechange	每个状态改变时都会触发这个事件处理器，通常会调用一个 JavaScript 函数
readyState	请求的状态
responseText	服务器的响应，表示为一个串
responseXML	服务器的响应，表示为 XML。这个对象可以解析为一个 DOM 对象
status	服务器的 HTTP 状态码(例如，200 对应 OK，404 对应 Not Found(未找到)等)
statusText	HTTP 状态码的相应文本(OK 或 Not Found 等)

下面重点对 XMLHttpRequest 对象的 readyState 属性进行介绍。根据它的值，可以得知 XMLHttpRequest 的执行状态，以便开发人员随之做出相应的处理。readyState 属性代码如表 12-2 所示。

表 12-2　readyState 属性代码

代　号	说　明
0	代表未初始化的状态。创建了一个 XMLHttpRequest 对象，但尚未初始化
1	代表连接状态。已经调用了 open 方法，并且已经准备好发送请求
2	代表发送状态。已经调用了 send 方法发出请求，但尚未得到响应结果
3	代表正在接收状态。已经接收了 HTTP 响应的头信息，正在接收响应内容
4	代表已加载状态。此时响应内容已完全被接收

此外，XMLHttpRequest 对象还提供了包括 send()和 open()在内的 6 个方法，用来向服务器发送 HTTP 请求，并设置相应的头信息。表 12-3 列出了 XMLHttpRequest 对象提供的方法及其简要说明。

表 12-3　XMLHttpRequest 对象的标准方法

名　称	说　明
abort()	中止当前请求
open(method,url)	使用请求方式(GET 或 POST 等)和请求地址 URL 以初始化一个 XMLHttpRequest 对象(这是该方法最常用的重载形式)
send(args)	发送数据，参数是提交的字符串信息
setRequestHeader(key,value)	设置请求的头部信息
getResponseHeader(key)	检索响应的头部值
getAllResponseHeaders()	返回响应头部信息(键/值对)的集合

12.1.2　实例描述

以前，人们需要用钱的时候，就会到银票上指定的钱庄去兑现，很不方便。现在，人们不需要受到这样的约束，只需要一张银行卡就可以取到现金。当然，不管是在哪个银行的自动取

款机上都可以。

使用 XMLHttpRequest 对象异步访问控制器动作，这个 XMLHttpRequest 对象就相当于银行卡，可以获取异地的信息。下面这个案例将讲解使用 Ajax 异步访问控制器动作。

12.1.3 实例应用

【例 12-1】异步访问控制器动作。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Ajax。

(2) 编写页面元素。除换行标记外，页面有两个 DOM 元素，功能分别为：按钮 Access 用于触发异步请求事件，browser 容器用于显示异步请求的页面执行结果，HTML 代码如下：

```
<input id="Access" type="button" value="Access" />
<br />
<div id="browser"></div>
```

(3) 在 Home 控制器中创建一个没有返回值的普通动作方法 Get_page，并用 Response.Write 输出静态网页，实现代码如下：

```
Response.Write("<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.1//EN'
'http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd'>");
Response.Write("<html>");
Response.Write("<head>");
…
Response.Write("</head>");
Response.Write("<body>");
Response.Write("<div class='container'>");
Response.Write("<div class='content'>");
Response.Write("<h1>中国古代的思想家认为:</h1>");
Response.Write("<p>孟子曰：人之初，性本善<a href='index.html'>荀子曰：人之初，性本
恶，善者伪也</a> 告子说：人之初，性本无向，犹如水，决之东则东，决之西则西.</p>");
Response.Write("<cite>人性是罪恶的 来到这世界是为了赎罪，可我们却在越陷越深.马克思说
认为：人是社会关系的产物.</cite>");
…
```

(4) 代码中创建了 4 个函数：CreateRequest 用于创建 XMLHttpRequest 对象；ResponseHandler 函数是一个回调函数，在通信成功并且返回结果正常的时候修改页面 DOM 的内容；AjaxAccess 是一个异步请求函数，在该函数中异步请求目标 URL；最后一个是页面加载事件处理程序，它为 Access 按钮绑定一个单击事件。

通过异步请求方式访问控制器动作，JavaScript 实现代码如下：

```
<script src="../../Scripts/jquery-1.4.1.min.js"
type="text/javascript"></script>
<script language="javascript" type="text/javascript">
   var xmlRequest;
   function CreateRequest() {
      /* 创建 XMLHttpRequest 对象 */
      if (window.ActiveXObject) {
         return new ActiveXObject("Microsoft.XMLHTTP");
      } else if (window.XMLHttpRequest) {
         return new XMLHttpRequest();
```

```
        }
    }
    function ResponseHandler() {
        if (xmlRequest.readyState == 4 && xmlRequest.status == 200) {
            /* 如果通信成功，并且响应正常，执行以下操作 */
            var reqContent = xmlRequest.responseText;
            document.getElementById("browser").innerHTML = reqContent;
        }
    }
    function AjaxAccess() {
        /* 异步请求 */
        xmlRequest = CreateRequest();                  //创建 XMLHttpRequest 对象
        xmlRequest.onreadystatechange = ResponseHandler;  //设置回调函数
        xmlRequest.open("GET", "/home/Get page");          //初始化请求对象
        xmlRequest.send(null);                         //发送请求信息
    }
    window.onload = function () {
        //设置按钮单击事件
        document.getElementById("Access").onclick = AjaxAccess;
    }
</script>
```

12.1.4　运行结果

运行该页面，单击 Access 按钮以后，待请求获取响应结果，div 中就会显示 Get_page 动作方法中输出的页面。运行效果如图 12-1 所示。



图 12-1　异步访问控制器动作

12.1.5 实例分析



源码解析

该案例使用 XMLHttpRequest 对象访问控制器动作，首先创建 XMLHttpRequest 对象，这是很关键的一步，实现代码如下：

```
if (window.ActiveXObject) {
   return new ActiveXObject("Microsoft.XMLHTTP");
} else if (window.XMLHttpRequest) {
    return new XMLHttpRequest();
}
```

接着通过对这个对象的方法或属性进行设置，以实现对控制器动作的访问。

12.2 使用 Ajax 获取数据

前面我们讲到以 POST 方式向页面发送数据，本节将讲解以 GET 方式获取数据。

**视频教学：**光盘/videos/12/12.2　使用 Ajax 获取数据　　　长度：8 分钟

12.2.1 基础知识——$.get()方法

$.get()方法用于以 GET 方式异步传递一些参数给服务器中的页面，该方法属于 jQuery 的全局方法。

$.get()方法的语法格式如下：

```
var xmlReq = $.get(url, [data], [callback], [type]);
```

其中，url 参数是一个字符串，表示异步请求的 URL 地址；data 参数用于指定执行异步请求时附加的参数列表，它们以“键/值”对的形式出现，是可选参数；callback 参数指定当响应成功时执行的回调函数，是可选参数；type 参数用于设置返回内容的格式，可选项有 xml、html、script、json、jsonp 和 text 等，默认值是 html。

$.get()方法的回调函数 callback 可以接受两个值，分别表示请求结果和响应状态。该回调函数的语法格式如下：

```
function(data,textStatus){
    /* data : 请求结果，可以是 html、xml、text 和 JSON 等 */
    /* textStatus : 响应状态，可能是 success 等 */
}
```

$.get()方法的返回值是它所创建的 XMLHttpRequest 对象。在大多数情况下，并不需要对它进行直接操作。不过，如果需要以手工方式取消请求，则会用到返回值。

例如，下面的示例代码演示了$.get()方法的使用情况。

```
$.get("check.aspx",                  /*  以 GET 方式请求 check.aspx  */
/*  将 name=somboy&pwd=123456 参数附加到 URL  */
{ name: "somboy", pwd: "123456" } ,
function(data){                      /*  设置回调函数  */
alert("返回结果为:  " + data);         /*  弹出响应结果  */
}
);
```

12.2.2　实例描述

关于页面的局部刷新，据了解，许多程序员都习惯用 Ajax 来实现。例如图书管理系统的前台首页，游客可以通过选择想要查找的书籍类型来进行页面的局部刷新，以便快速找到你想要的书籍。本节将以 GET 请求页面的方式讲解使用 Ajax 如何实现局部刷新。

12.2.3　实例应用

【例 12-2】使用 Ajax 获取数据。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Page_get。

(2) 在页面中有一个下拉列表 blogClass；一个 div 标签，用于接收下拉列表项所对应的数据；一个按钮，用于响应请求。代码如下：

```
<select id="blogClass">
<option selected="selected">所有</option>
<option>CSS</option>
<option>C#</option>
</select>
<input type="button" id="Search" value="Search"/>
<div id="blogList" style=" margin-top:20px"></div>
```

(3) 选择下拉列表中的任何一个选项，单击 Search 按钮就可以显示出对应的数据。例如，当你选择 C#选项的时候，div 标签中就会显示“C#面向对象”，jQuery 实现代码如下：

```
<script src="../../Scripts/jquery-1.4.1.min.js"
type="text/javascript"></script>
<script language="javascript" type="text/javascript">
   $(document).ready(function () {
      $("#Search").click(function () {
         /* 使用 Get 方式请求指定 URL */
         $.get("/home/Get data",
        {
           key: $("#blogClass").val()
        },
        function (data) {
           $("#blogList").html(data);
        });
      });
      $("#Search").click();  //触发一次单击事件
   });
```

```
</script>
```

(4) 在 Home 控制器下创建一个没有返回值的方法 Get_data，实例化键值对的类 blogClass，并为其添加数据，实现代码如下：

```
public void Get_data()
{
    Dictionary<string, string> blogClass = new Dictionary<string, string>();//
实例化 blogClass 类
    blogClass.Add("CSS", "CSS 中的 padding");                    //添加键值对
    blogClass.Add("C#", "C#面向对象");
    blogClass.Add("所有","CSS 中的 padding, C#面向对象");
    string key = Request["key"];                                //获取请求服务器的关
键字
    foreach (string str2 in blogClass.Keys)                     //遍历键的集合
    {
        if (str2 == key)
        {
            Response.Write(blogClass[key]);                     //输出键值
        }
    }
}
```

12.2.4 运行结果

运行程序，选择 C#选项，再单击 Search 按钮，页面局部刷新，效果如图 12-2 所示。

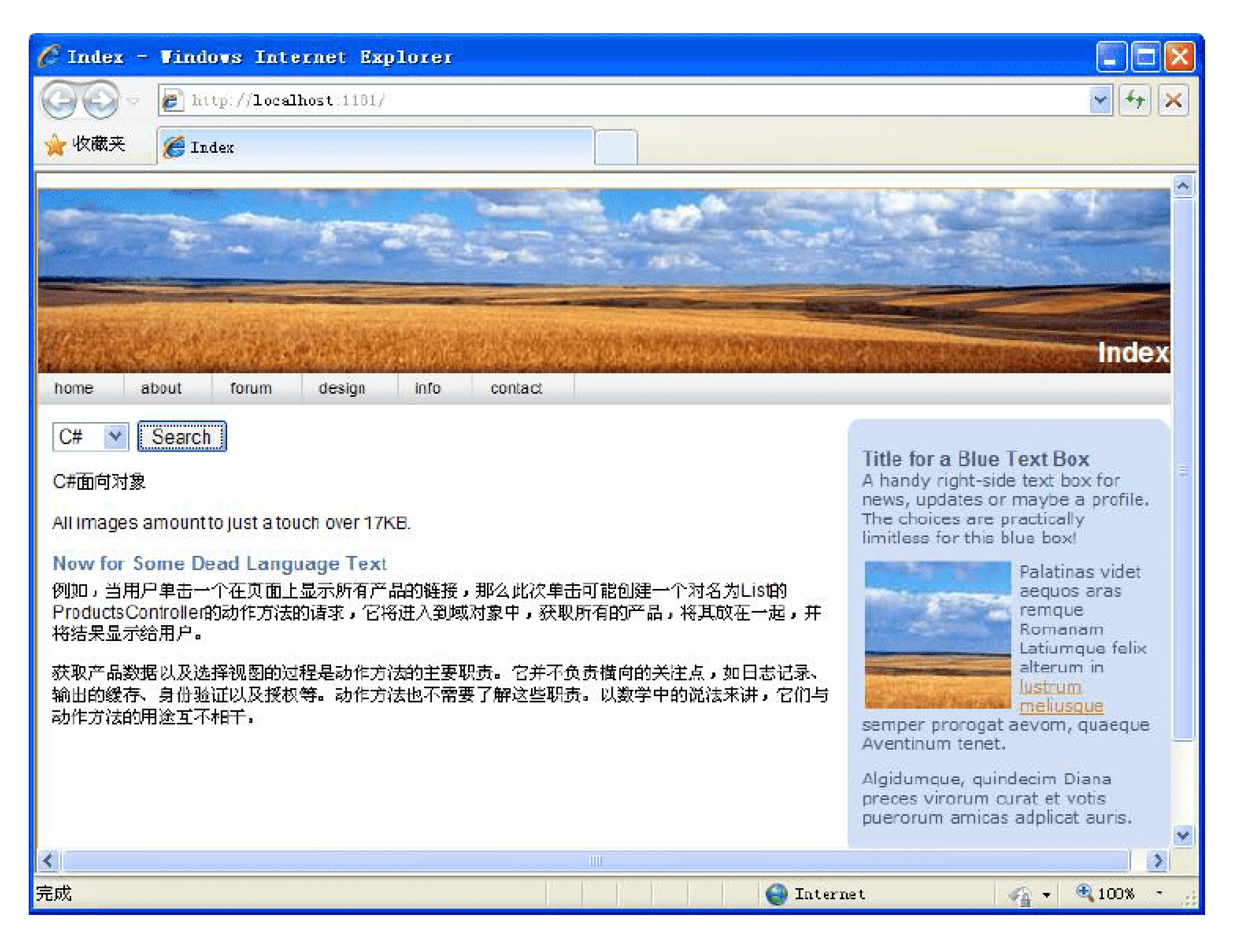


图 12-2　使用 Ajax 获取数据

12.2.5　实例分析

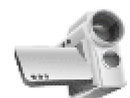


源码解析

该案例为页面 Search 按钮绑定了一个单击事件处理程序，在该事件处理程序中，以 GET 方式请求指定 URL，并为其传递参数。请求完成将请求结果填充到页面的 div 中。最后直接触发一次单击事件，获得页面初始内容。

12.3　使用 Ajax 向页面发送数据

Ajax 具有异步提交的功能，可以使页面局部刷新，而不用将整个页面刷新。本节将讲解 ASP.NET MVC 框架中 Ajax 以 POST 方式发送数据的功能。



视频教学： 光盘/videos/12/12.3　使用 Ajax 向页面发送数据　　长度：7 分钟

12.3.1　基础知识——$.post()方法

与$.get()方法类似，$.post()方法也是 jQuery 全局方法，它能够实现以 POST 方式异步提交数据到服务器端。

$.post()方法的语法格式如下：

```
var xmlReq = $.post(url, [data], [callback], [type])
```

可以看到，该方法的语法结构和$.get()方法一样，而且用法和参数列表及返回值说明都完全一致。

$.post()方法的语法结构如下：

```
var xmlReq = $.post(url, [data], [callback], [type])
```

可以看到，该方法的语法结构和$.get()方法一样，页且用法和参数列表及返回值说明都完全一致。

但是，即然 jQuery 提供了这个方法，就说明它还是有存在的必要之处的。总体来说，它们有以下几点不同：

- 在服务器端可能会为同一个 URL 的不同请求方式进行不同的操作(特别是在 MVC 模式下的应用程序中更为常见)，所以不同的请求方式可能会被服务器响应出不同的结果。
- $.get()方法是以 GET 方式提交的数据，所有的参数信息都将追加到 URL 后面。而 Web 请求一般对 URL 长度有一定限制，所以$.get()方法传递的数据长度也有一定上限，而$.post()方法是将参数作为消息的实体发送到服务器的，对数据无长度上的影响

- 由于浏览器的缓存功能会把所有请求的 URL 进行临时存储，所以以$.get()方法追加到 URL 中的数据也会被浏览器保存到磁盘上，这样如果参数是比较机密的数据，则可能会带来安全隐患，而作为数据内容的$.post()方法则不存在这个问题。

12.3.2 实例描述

网上购物已经成为当今社会的时尚，这样省时省事又省力，具有三重功效。例如淘宝网，里面有很多我们日常生活需要的东西，我想看看数码产品这个模块，那么页面将只刷新数码模块的产品信息，其他数据不变。这样，一来可以提高用户网上冲浪的速度，二来可以节省很多资源。

下面这个案例将讲解如何使用 Ajax 向页面发送数据以实现局部刷新的效果。

12.3.3 实例应用

【例 12-3】使用 Ajax 向页面发送数据。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Page_send。

(2) 以 POST 方式访问页面，并向其传送数据。在 index 视图中，编写如下 jQuery 实现代码：

```
<script src="../../Scripts/jquery-1.4.1.min.js" type="text/javascript">
</scr ipt>
<script language="javascript" type="text/javascript">
    $(document).ready(function () {
        $("#btn").click(function () {
            $.post('/home/Page_to', {
                text: 'my string',          //设置 text 属性值
                number: 23                  //设置 number 属性值
            },
            function (data) {
                alert('Your data has been saved.');
                $("#result").text(data);   //用 div 接收返回的数据
            });
        });
    })
</script>
```

(3) 在 Home 控制器里面创建一个没有返回值的方法 Page_to()，接收以上代码中设置 text 属性值，并输出结果，实现代码如下：

```
public void Page_to()
{
    string str = Request["text"];
    Response.Write("返回结果是： "+str);
}
```

(4) 在 index 中添加一个客户端的提交按钮和一个 div 标签，其中 div 标签用于接收数据，

其代码如下：

```
<input id="btn" type="submit" value="发送数据"/>
```

12.3.4　运行结果

运行程序，单击【发送数据】按钮，效果如图 12-3 所示。

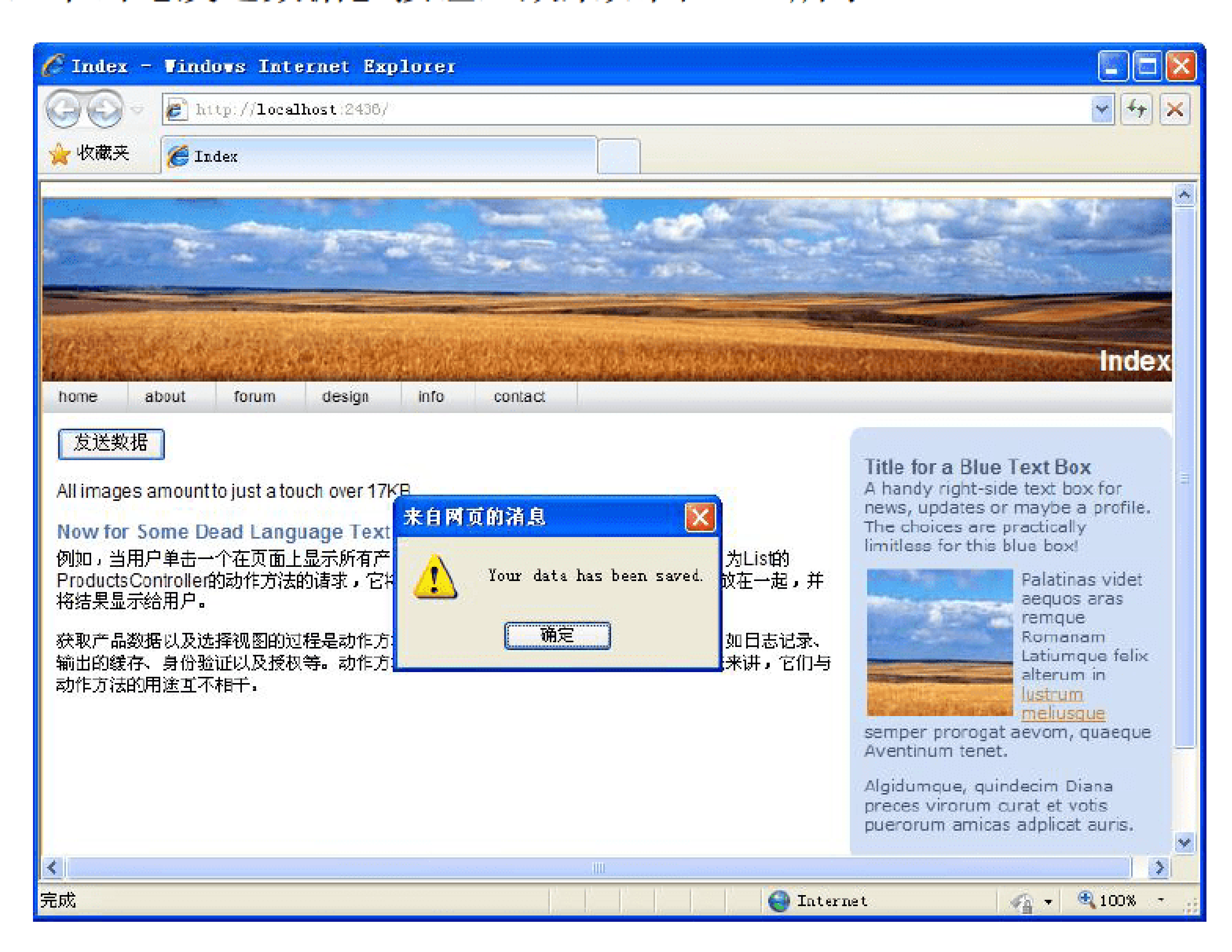


图 12-3　使用 Ajax 向页面发送数据

单击【确定】按钮，页面将被局部刷新，效果如图 12-4 所示。



图 12-4　接收数据

12.3.5 实例分析



源码解析

该案例在单击【发送数据】按钮的事件中设置了 text 属性的值，并且以 POST 方式提交到 Page_to 方法中。Page_to 方法来接收 text 属性值，并输出结果。在 jQuery 代码中用 div 标签来获取返回的数据，那么 index 页面将局部刷新，并显示返回结果，代码如下：

```
function (data) {
   alert('Your data has been saved.');
   $("#result").text(data);
});
```

12.4 异步读取书籍名称

在其他页面或者方法中，读取已保存的书籍的名称。在 ASP.NET MVC 框架中读取不同方法中保存的数据，可以使用 Ajax 的异步请求。前面讲到使用 Ajax 对 URL 进行 POST 和 GET 等的请求方式。本节将讲解$.ajax()方法的使用。



视频教学：光盘/videos/12/12.4 异步读取书籍名称 长度：11 分钟

12.4.1 基础知识——$.ajax()方法

$.ajax()方法属于 jQuery 对 Ajax 提供的最底层方法，前面介绍的$.get()和$.post()方法均属于更高一级的实现。

$.ajax()方法的语法很简单，如下所示：

```
$.ajax(options);
```

该方法只有一个 options 参数，该参数是一组"键/值"对集合，通过这些"键/值"来对 Ajax 请求的各个选项进行配置，像请求方式、请求 URL 或者回调函数等。

options 参数的所有选项都是可选的，表 12-4 中列出了该参数所有可能的选项及其说明。

表 12-4 $.ajax()方法的 options 参数

名 称	说 明
async	指定是否以异步方式发送请求，默认为 true
beforeSend	指定发送请求之前要调用的函数
cache	指定是否强制浏览器缓存被请求的页面，默认为 true
complete	请求完成后的回调函数，无论请求成功与否，都将触发该回调函数。该回调函数可以有两个参数，第一个参数用于访问当前使用的 XMLHttpRequest 对象，第二个参数返回当前请求的执行状态(success 或 error 等)

续表

名　称	说　明
contentType	发送信息至服务器的内容编码类型。默认值为 application/x-www-form-urlencoded
data	发送到服务器的数据。该属性可以是对象或字符串，如果是对象，则自动转换为字符串
dataType	预期服务器返回的数据类型，该属性为字符串类型。可选值有 xml、html、script、json、jsonp 或者 text。如果没有设置该项，jQuery 将根据 HTTP 包中的 MIME 信息来进行智能判断
error	请求执行失败时的回调函数，该回调函数可以有 3 个参数，分别是 XMLHttpRequest 对象，描述执行状态的字符串，以及捕获的错误对象
global	是否触发全局 Ajax 事件，默认值为 false
ifModified	指定是否仅在服务器数据改变时获取新数据，默认为 false
jsonp	用于重写 jsonp 请求中的回调函数名称，默认值为 callback
password	指定响应 HTTP 访问认证请求的密码
processData	是否将发送的数据转换为对象以配合默认内容类型 application/x-www-form-urlencoded，默认为 true
scriptCharset	只有当请求时 dataType 为 jsonp 或 script，并且 type 是 GET 才会用于强制修改 charset
success	请求执行成功时的回调函数。该回调函数有两个参数，第一个参数用于访问该请求返回的数据，第二个是描述执行状态的字符串
timeout	设置请求超时时间，以毫秒为单位
type	请求方式(如 GET、POST 等)，默认为 GET
url	发送请求的地址
username	指定用于响应 HTTP 访问认证请求的用户名

下面来看一些使用$.ajax()方法进行异步交互的例子。以下的代码将以 GET 方式加载 content.js 文件：

```
$.ajax({
  type: "GET",
  url: "content.js",
  dataType: "script"
});
```

再看一个例子，即如何使用$.ajax()方法发送 POST 方式的异步请求，并传递 URL 参数和使用回调函数。代码如下：

```
$.ajax({
   type: "POST",
   url: "check.aspx",
   data: "name=somboy&pwd=123456",
   success: function(msg){
     alert( "result:" + msg );
   }
});
```

12.4.2　实例描述

Ajax 最大的优点就是能够局部刷新，前面我们已经讲了几种关于 Ajax 的方法，下面这个案例将为大家讲解 Ajax 的另一种使用方法。

12.4.3　实例应用

【例 12-4】异步读取书籍名称。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 AjaxMvc。

(2) 在 index 视图中，有一个按钮和一个 div 标签，代码如下：

```
<input id="Access" type="button" value="Access" />
<div id="browser" style=" margin-top:20px"></div>
```

(3) 在 Home 控制器下创建一个没有返回值的方法 Get_page。实例化一个泛型类 lst，并为其添加数据，然后遍历输出，实现代码如下：

```
public void Get_page()
{
    List<string> lst = new List<string>();
    lst.Add("C#编程基础");
    lst.Add("C#高级编程");
    lst.Add("C#入门");
    foreach (string str in lst)
    {
      Response.Write(str+"<br />");
    }
}
```

(4) 使用$.ajax()方法异步请求 Home 控制器下的 Get_page 方法，如果请求成功，将执行一个事件的处理，用 div 标签接收返回的数据，实现代码如下：

```
<script src="../../Scripts/jquery-1.4.1.min.js"
type="text/javascript"></script>
<script language="javascript" type="text/javascript">
    $(document).ready(function () {
        $("#Access").click(function () {
            $.ajax({
                type: 'GET',
                url: '/home/Get_page',
                success: function (data) {
                    $("#browser").text(data);
                }
            });
        });
    });
</script>
```

12.4.4　运行结果

运行程序，单击 Access 按钮，效果如图 12-5 所示。

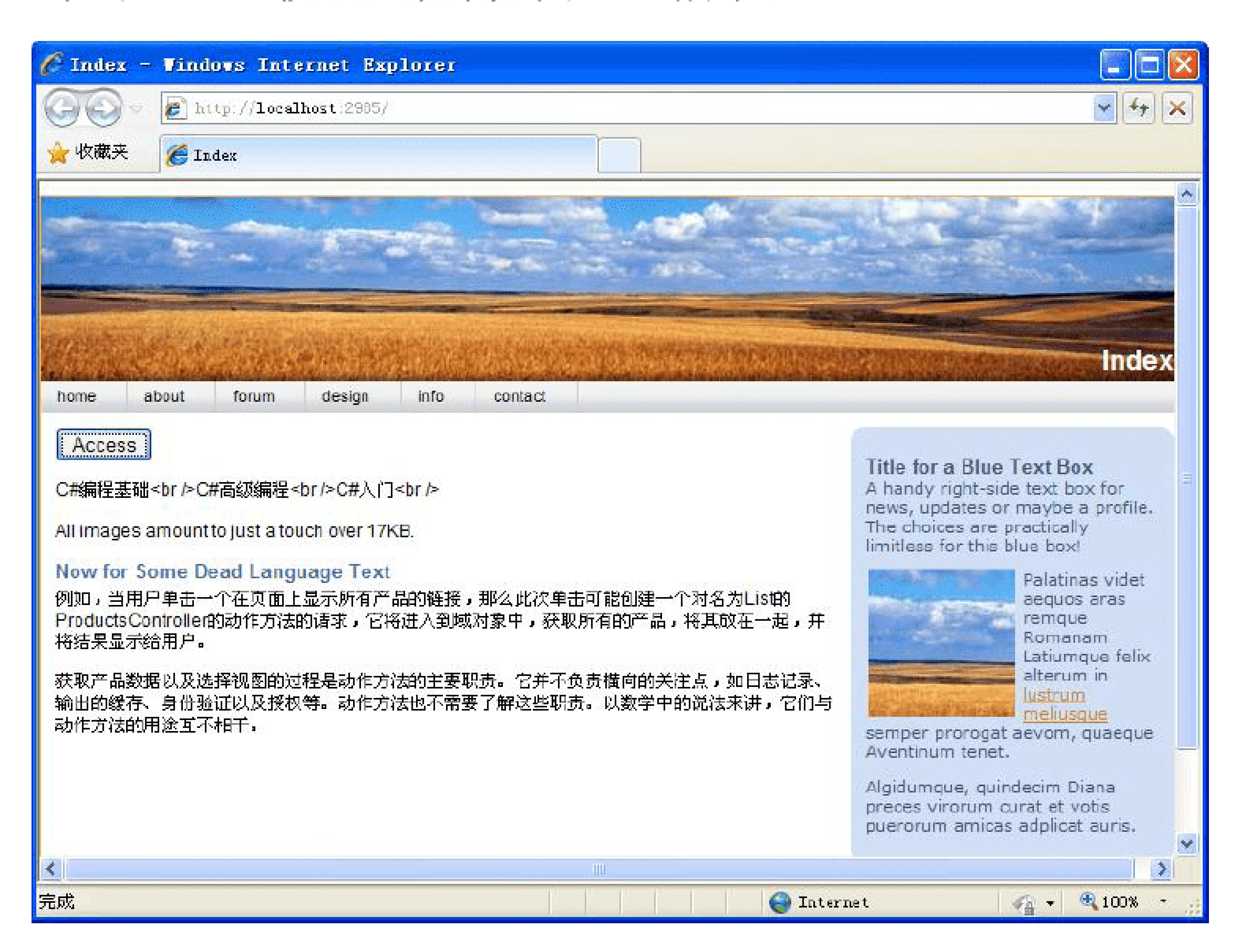


图 12-5　异步读取书籍名称

12.4.5　实例分析



源码解析

为 Home 控制器创建了一个没有返回值的方法 Get_page()，用于响应 Ajax 的请求，这个方法保存了关于书籍名称的数据。当 Ajax 请求处理完毕，将返回这些数据，最后将这些数据显示在 index 视图中的 div 标签里，这样就实现了页面的局部刷新，也节省了内存的使用。

12.5　异步请求 JSON 数据

异步请求控制器动作方法中返回的 JSON 数据。JSON(JavaScript Object Notation)是一种轻量级的数据交换格式，易于阅读和编写，同时也易于机器解析和生成。它基于 JavaScript(Standard ECMA-262 3rd Edition - December 1999)的一个子集。JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯(包括 C、C++、C#、Java、JavaScript、Perl 和 Python 等)。这些特性使 JSON 成为理想的数据交换语言。



视频教学：光盘/videos/12/12.5　异步请求 JSON 数据　　长度：7 分钟

12.5.1 基础知识——$.getJSON()方法

JSON 是一种轻量级的数据交换格式，非常适合于服务器与 JavaScript 的交互。和 XML 一样，JSON 也是基于纯文本的数据格式，而且 JSON 比 XML 还简单。

例如，一个学生的基本信息包含学号(studentNumber)、姓名(name)、年龄(age)和性别(isMale)等。用 JSON 表示一个学生集合，代码如下：

```
[{
    "studentNumber" : "HNHZ11011240",
    "name" : "lee xiao long",
    "age" : 20,
    "isMale" : true
},{
    "studentNumber" : "HNHZ10053532",
    "name" : "zhao hao tai",
    "age" : 24,
    "isMale" : true
}]
```

代码中以方括号括起来的为集合，在 JavaScript 中像数组一样可被直接访问。花括号括起来的是对象，可以直接操作对象里的属性。

$.getJSON()方法是 jQuery 中的一个全局方法，该方法可以通过 GET 的方式请求载入 JSON 数据。其语法如下：

```
var xmlReq = $.getJSON(url, [data], [callback])
```

该方法的 3 个参数和$.get()方法一样，这里就不做说明了。

在 ASP.NET MVC 中，通常我们会访问一个保存 JSON 数据的方法，这个方法的返回值类型是 JsonResult，以下介绍了关于 JSON 的几个重载方法。

- Json(Object) 创建一个将指定对象序列化为 JavaScript 对象表示法(JSON)的 JSONResult 对象(继承自 Controller)。
- Json(Object, String) 创建一个将指定对象序列化为 JavaScript 对象表示法(JSON)格式的 JsonResult 对象(继承自 Controller)。
- Json(Object, JsonRequestBehavior) 创建 JsonResult 对象，该对象使用指定的 JSON 请求行为将指定对象序列化为 JavaScript 对象表示法(JSON)格式(继承自 Controller)。
- Json(Object, String, Encoding) 创建一个将指定对象序列化为 JavaScript 对象表示法(JSON)格式的 JsonResult 对象(继承自 Controller)。
- Json(Object, String, JsonRequestBehavior) 创建 JsonResult 对象，该对象使用指定内容类型和 JSON 请求行为将指定对象序列化为 JavaScript 对象表示法(JSON)格式(继承自 Controller)。
- Json(Object, String, Encoding, JsonRequestBehavior) 创建 JsonResult 对象，该对象使用内容类型、内容编码和 JSON 请求行为将指定对象序列化为 JavaScript 对象表示法(JSON)格式(继承自 Controller)。

12.5.2　实例描述

在 Web 项目开发中，保存数据一般会用泛型等集合列表。下面这个案例让我们就一起来学习如何异步请求 JSON 数据，当然 JSON 里面保存的也是集合列表，只不过返回的是 JSON 数据。

12.5.3　实例应用

【例 12-5】异步请求 JSON 数据。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Ajax_json。

(2) 在 Home 控制器下创建一个返回 JSON 数据的方法 Get_json，并将返回的 JSON 设置为允许 GET 访问，实现代码如下：

```
public JsonResult Get json()
{
   IList<object> list = new List<object>()
   {
      new { name="zhangsan", age=23, isMale=true },
      new { name="jo", age=34, isMale=true },
      new { name="obama", age=54, isMale=true }
   };
   return Json(list,JsonRequestBehavior.AllowGet);
}
```

(3) 在 index 视图中添加一个表格，用于显示请求获得的 JSON 数据，实现代码如下：

```
<table id="people" cellspacing="1">
<thead>
<tr><td>Name</td><td>Age</td><td>Sex</td></tr>
</thead>
<tbody></tbody>
</table>
```

(4) 编写 jQuery 代码，使用 getJSON()方法异步请求 Get_json 方法中的 JSON 数据，并将数据显示在表格中，实现代码如下：

```
<script src="../../Scripts/jquery-1.4.1.min.js"
type="text/javascript"></script>
<script language="javascript" type="text/javascript">
   $(document).ready(function () {
      /* 异步请求，载入 JSON 数据 */
      $.getJSON("/home/Get json",
       function (data) {
          /* 遍历请求结果 */
          $.each(data,
              function (index, p) {
                 var html = "<tr><td>" + p.name + "</td><td>" + p.age +
                  "</td><td>" + (p.isMale ? "male" : "female") + "</td></tr>"
                 $("tbody").append(html);
              });
```

```
        });
    });
</script>
```

12.5.4 运行结果

运行程序，效果如图 12-6 所示。

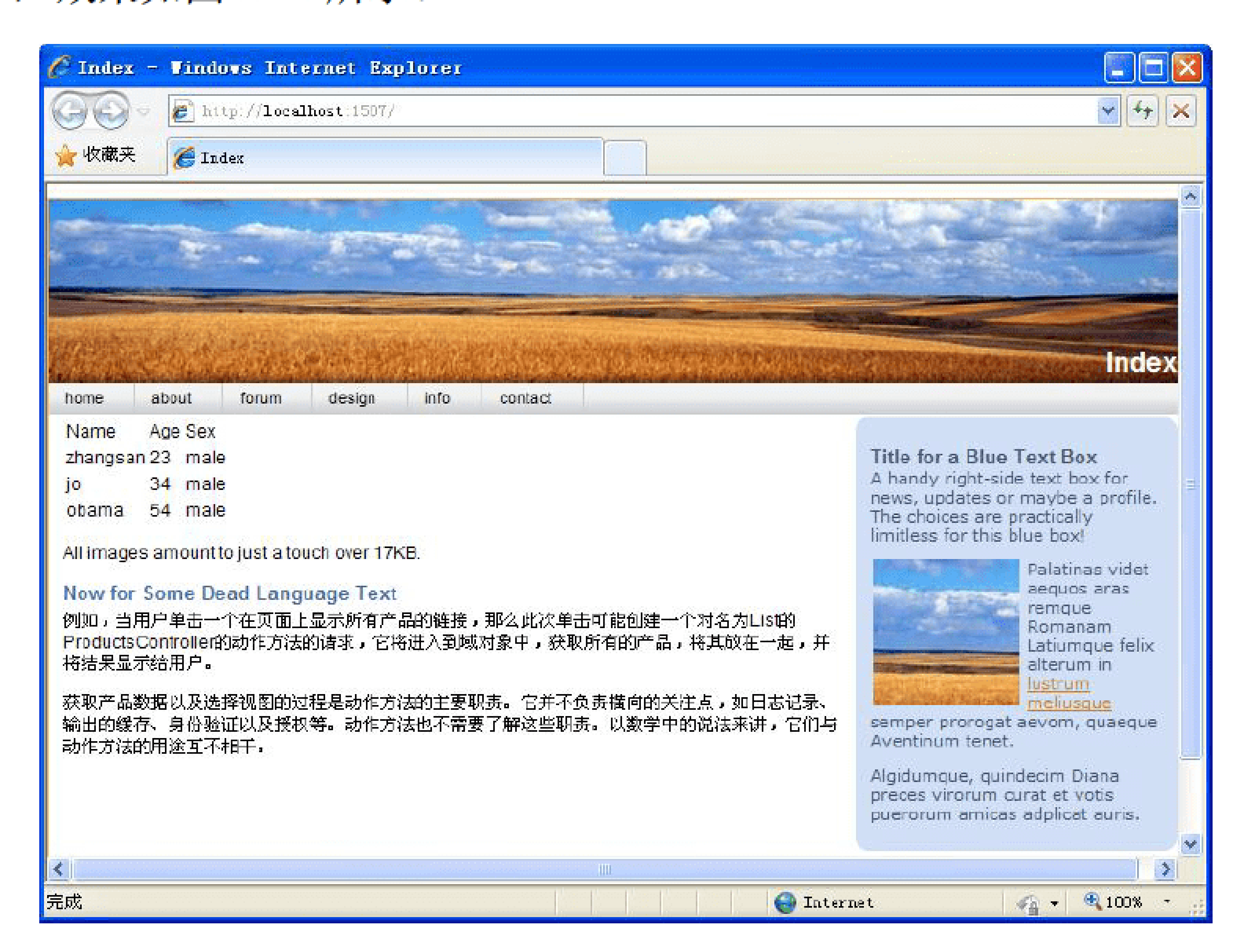


图 12-6 异步请求 JSON 数据

12.5.5 实例分析



源码解析

在 Home 控制器下创建一个返回类型为 JsonResult 的动作方法 Get_json，将 list 泛型集合以 JSON 数据的形式返回，实现代码如下：

```
return Json(list,JsonRequestBehavior.AllowGet);
```

其中，JsonRequestBehavior.AllowGet 表示允许以 GET 方式访问。最后在页面中用 getJSON()方法获取 JSON 数据。

12.6 提交 Ajax 表单

通常，我们将想要提交的数据放在 Form 表单中，本节将讲解如何提交 Ajax 表单中包含的数据。需要注意的是，一定要引用 ASP.NET MVC 框架自带的两个脚本文件 MicrosoftAjax.js 和 MicrosoftMvcAjax.js。

视频教学：光盘/videos/12/12.6　提交 Ajax 表单　　　　　长度：7 分钟

12.6.1　基础知识——Ajax.BeginForm()方法

ASP.NET MVC 有第三种使用 Ajax 的方式——Ajax Helper，它包含以下 4 个方法。

- Ajax.ActionLink()　渲染成一个超链接的标签，类似于 Html.ActionLink()。当它被单击之后，将获取新的内容并插入 HTML 页面中。
- Ajax.BeginForm()　渲染成一个 HTML 的 Form 表单，类似于 Html.BeginForm()。当它提交之后，将获取新的内容并插入 HTML 页面中。
- Ajax.RouteLink()　类似于 Ajax.ActionLink()。不过它可以根据任意的 routing 参数生成 URL，而不必包含调用的 action。使用最多的场景是自定义的 IController，里面没有 action。
- Ajax.BeginRouteForm()　类似于 Ajax.BeginForm()，这个 Ajax 等同于 Html.RouteLink()。

下面具体了解 Ajax.BeginForm()的参数，如表 12-5 所示。

表 12-5　Ajax.BeginForm()的参数

参　数	说　明
actionName	将处理请求的操作方法的名称
controllerName	将处理请求的控制器的名称
ajaxOptions	提供异步请求选项的对象

其中，ajaxOptions 对象包含以下几个可选属性，如表 12-6 所示。

表 12-6　ajaxOptions 对象的属性

属　性	说　明
HttpMethod	获取或设置 HTTP 请求方法(POST 或 GET)
UpdateTargetId	获取或设置要使用服务器响应来更新的 DOM 元素的 Id
InsertionMode	获取或设置指定如何将响应插入目标 DOM 元素的模式
Confirm	获取或设置在提交请求之前，显示在确认窗口中的消息
LoadingElementId	获取或设置在加载 Ajax 函数时，要显示的 HTML 元素的 Id
OnBegin	获取或设置在更新页面之前调用的 JavaScript 函数的名称
OnComplete	获取或设置在实例化响应数据之后但更新页面之前，要调用的 JavaScript 函数
OnFailure	获取或设置在页面更新失败时，要调用的 JavaScript 函数
OnSuccess	获取或设置在成功更新页面之后，要调用的 JavaScript 函数
Url	获取或设置要向其发送请求的 URL

12.6.2 实例描述

某天，我看到一个很不错的论坛，想下载文件，结果还得注册。没办法，现在这个社会，做什么都得注册。找工作要注册，发表微博要在微博上注册账号等，最普遍的就是结婚要先注册，这是规矩。

那么这些所谓的注册页面，可以用很多种语言去实现，在下面这个案例中，我们就用 Ajax 来实现。

12.6.3 实例应用

【例 12-6】提交 Ajax 表单。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 AjaxApp。

(2) 页面中有一个文本框和一个提交按钮，通过用户单击按钮，得到搜索结果，更新页面上的 div 标签里的文本值。创建一个 Ajax 的表单(使用 Microsoft Ajax 库，该库与 ASP.NET MVC 在一起)，方法是使用 ASP.NET MVC 的辅助方法，实现代码如下：

```
<script src="../../Scripts/MicrosoftAjax.js"
type="text/javascript"></script>
<script src="../../Scripts/MicrosoftMvcAjax.js"
type="text/javascript"></script>
<%using (Ajax.BeginForm("helloajax", new AjaxOptions { UpdateTargetId =
"results" }))
  { %>
  <%=Html.TextBox("query",null,new {size=40}) %>
  <input type="submit" />
<%} %>
<div id="results"></div>
```

以上代码引用了 MicrosoftAjax.js 和 MicrosoftMvcAjax.js 两个脚本文件。

(3) 在 Home 控制器里创建一个简单的动作，该动作将呈现搜索字符串作为表单的结果，实现代码如下：

```
public string helloajax(string query)
{
    return "你输入的是：" + query;
}
```

12.6.4 运行结果

运行程序，在文本框中输入任意字符串，单击【提交查询内容】按钮，div 标签里面的文本值将更新为之前输入的字符串，效果如图 12-7 所示。

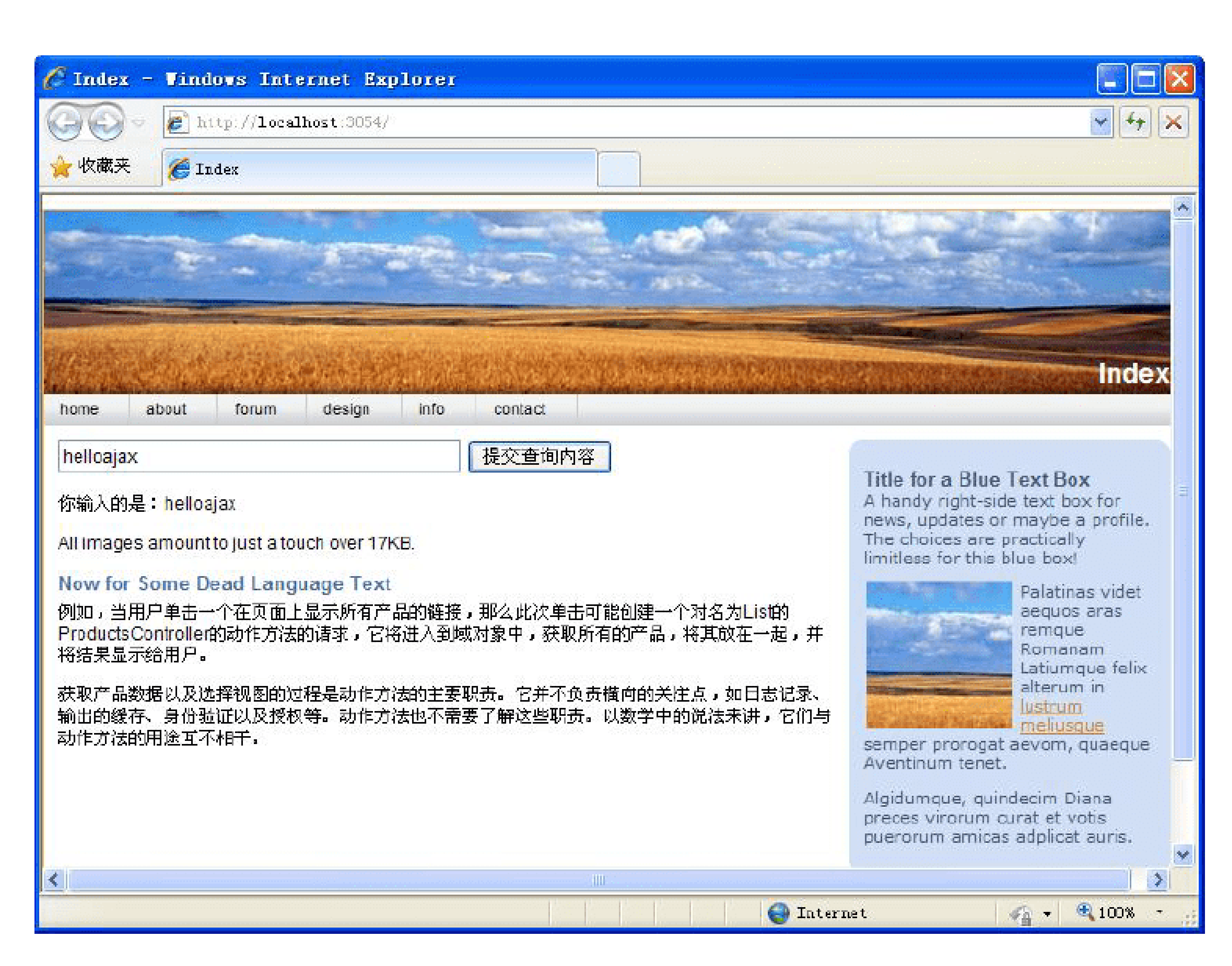


图 12-7　提交 Ajax 表单

12.6.5　实例分析

**源码解析**

该案例主要讲述了如何提交 Ajax 表单，在 ASP.NET MVC 框架中，自动为用户提供了关于 Ajax 的脚本文件，这里用到的两个脚本文件分别是 MicrosoftAjax.js 和 MicrosoftMvcAjax.js。在 Ajax 表单中有一个文本框和一个提交按钮，如何成为一个 Ajax 表单呢？关键在于 Ajax.BeginForm()方法的使用。

12.7　获取当前时间

获取当前时间，用 DateTime.Now 即可获取，很简单。然而，我们想通过 Ajax 异步请求，获取它的周期是怎么样的。本节主要讲解有关 Ajax 的全局事件。

**视频教学：**光盘/videos/12/12.7　获取当前时间　　长度：8 分钟

12.7.1　基础知识——Ajax 全局事件

前面我们介绍过，使用$.ajax()方法可以设置一些回调函数，它们将在请求发送之前，请求完成后，以及请求成功或者失败时执行。使用这些方式设置的回调函数，仅适用于某次 Ajax 请求。若要对 Ajax 请求设置全局回调函数，则需要对 Ajax 全局事件进行绑定。

jQuery 提供的全局事件函数主要有如下几个。

- ajaxComplete(callback)　Ajax 请求完成时触发该事件。
- ajaxError(callback)　Ajax 请求出现错误时触发该事件。
- ajaxSend(callback)　Ajax 请求发送前触发该事件。
- ajaxStart(callback)　Ajax 请求开始时触发该事件。
- ajaxStop(callback)　Ajax 请求结束时触发该事件。
- ajaxSuccess(callback)　Ajax 请求成功时触发该事件。

以上几个函数的参数 callback 都是触发事件时执行的事件处理程序，其中 ajaxStart()和 ajaxStop()方法的事件处理程序是一个无参的函数。其余的都可以有 3 个参数，格式如下：

```
function(event, XHR, settings){
    /* event 是触发的事件对象 */
    /* XHR 是 XMLHttpRequest 对象 */
    /* settings 是 Ajax 请求配置参数 */
}
```



当我们使用 jQuery 的 Ajax 方法时，不管是$.ajax()、$.get()或$.getJSON()等都会默认触发全局事件。只是通常不绑定全局事件，但实际上这些全局事件非常有用。

12.7.2　实例描述

在 Web 项目开发中，页面中存在多个甚至为数不少的 Ajax 请求，这些 Ajax 请求都有相同的消息机制。在 Ajax 请求开始前显示一个提示框，提示“正在读取数据”；Ajax 请求成功时提示框显示“数据获取成功”；Ajax 请求结束后则隐藏提示框。

12.7.3　实例应用

【例 12-7】获取当前时间。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 AjaxEvent。

(2) 在 index 视图中，有一个 div 标签，用于表现全局事件的执行过程；一个文本域，用于显示请求后得到的数据；一个按钮，用于触发全局事件。HTML 代码如下：

```
<div id="loading" style=" margin-top:20px;">Loading.....</div>
<textarea id="content"></textarea>
<input type="button" id="btn_loading" value="load" />
```

(3) 在 Home 控制器下创建一个返回值类型是 string 的动作方法 Get_date，用于返回当前时间，其实现代码如下：

```
public string Get_date()
{
    return DateTime.Now.ToString();
}
```

(4) 通过编写 jQuery 代码来展示全局事件的执行过程，其实现代码如下：

```
<script src="../../Scripts/jquery-1.4.1.min.js"
type="text/javascript"></script>
<script language="javascript" type="text/javascript">
    $(document).ready(function () {
        $("#loading").hide();
        $("#loading").ajaxStart(function () {//Ajax 请求开始时触发该事件
            $(this).show();                    //id 为 loading 的 div 标签显示
        });
        $("#loading").ajaxStop(function () {//Ajax 请求结束时触发该事件
            $(this).hide();                        //id 为 loading 的 div 标签隐藏
        });
        $("#btn loading").click(function () {//单击 loading
            $.get("/home/Get_date", null, function (data) {
                $("#content").text(data);
            });
        });
    });
</script>
```

12.7.4　运行结果

运行程序，单击 load 按钮，由于触发事件后持续的时间很短，所以看不到 loading…标记，效果如图 12-8 所示。



图 12-8　获取当前时间

12.7.5　实例分析



源码解析

该案例中，ajaxStart 事件发生在请求开始的时候，ajaxStop 事件发生在请求结束的时候。

当请求开始的时候，指定 div 标签显示；当请求结束的时候，指定 div 标签就会隐藏。由于这里请求的路径是本地路径，也就是 Home 控制器下的 Get_date 方法，事件从开始到结束的时间间隔非常短，所以执行效果很快显示出来。

12.8 常见问题解答

12.8.1 使用 Ajax 更新页面信息



使用 Ajax 更新页面信息的问题。
网络课堂：http://bbs.itzcn.com/thread-11077-1-1.html

我想在 A 页面触发一个事件，使用 Ajax 从数据库中取出 id=1 的姓名和性别，放在 A 页面的两个 input 里，请问该怎么实现，最好使用 jQuery。

【解决办法】比如你的页面上有两个 input。

```
<input id="userName" />
<input id="userSex" />
```

你可以使用如下方法异步请求服务器。

```
$(function(){
    $.getJSON(
        "Test.html",  //接收请求的服务器路径
        {
            id : 1
        },
        function(jsonData){
            $("#userName").val(jsonData.username);
            $("#userSex").val(jsonData.usersex);
        })
    $(document).ajaxError(function(){ alert("asdfasd"); });
});
```

之后你在服务器接收请求的 id 值，读取相应的记录，向客户端返回这样一段 JSON 文本即可。

```
{
    "username" : "zhangsan",
    "usersex" : "male"
}
```

12.8.2 使用 Ajax 的 getJSON()方法没反应



我使用 Ajax 的 getJSON()方法没反应，为什么？
网络课堂：http://bbs.itzcn.com/thread-11078-1-1.html

我使用 Ajax 的 getJSON()方法时没反应，为什么？

我有如下一段待处理的 JSON 数据。

```
{
    username : "李一",
    title : "测试 1",
    content : "这是测试 1 的内容"
}
```

在使用下面代码处理的时候没有反应。

```
$.getJSON(
    "json.html", null,
    function(json){
        $("#userName").val(json.username);
        $("#title").val(json.title);
        $("#content").val(json.content);
    })
```

什么原因啊？

【解决办法】应该是你的 JSON 数据属性名没有加引号所致。

这样修改一下：

```
{
    "username" : "李一",
    "title" : "测试 1",
    "content" : "这是测试 1 的内容"
}
```

这样就可以了。

12.8.3　为什么执行了 jQuery 中的 Ajax 还要刷新页面



为什么执行了 jQuery 中的 Ajax 还要刷新页面啊？

网络课堂：http://bbs.itzcn.com/thread-11079-1-1.html

为什么执行了 jQuery 中的 Ajax 还要刷新页面啊？

我的页面代码如下：

```
<form>
<input id="userName" />
<input id="userSex" />
<button id="submit">Submit</button>
</form>
```

jQuery 代码如下：

```
$(function(){
    $("#submit").click(function(){
        $.get( "test.html", null,
        function(){ /* Function Body */ });
    });
});
```

【解决办法】这不奇怪，单击按钮的时候触发了表单的提交事件。你在表单提交事件处理

程序中返回一个 false 就行了。代码如下：

```
$("form").submit(function(){ return false; });
```

把这行代码添加到 jQuer 代码的 ready 事件中就行了。

12.8.4 关于 ASP.NET MVC BeginForm 的问题



关于 ASP.NET MVC BeginForm 的问题。
网络课堂：http://bbs.itzcn.com/thread-11080-1-1.html

让我一直很困惑的是，为什么不直接用<form></form>？

【解决办法】Action 的 URL 可能受 UrlRouting 的影响而改变。

比如你将{controller}/{action} 规则改为{action}/{controller}.html。

是不是要将所有的页面的<form action="data/get"改为<form action="get/data.html"呢？

如果使用 Helper 就解决了这个问题。

其实所有的 Helper 无非就是解决程序中有变化的东西，比如绑定数据，或 UrlRouting 的 URL 或相对路径，如果你的地址一直不会变化，完全可以使用 HTML 标签。

12.9 习　题

一、填空题

(1) Ajax 的异步操作是由__________对象实现的。
(2) XMLHttpRequest 对象是浏览器中实现使用 HTTP 协议与__________交换数据的对象。
(3) 使用 GET 方式来进行异步请求的方法是__________。
(4) jQuery 实现 Ajax 的最底层的方法是__________。
(5) 异步请求 JSON 数据的使用方法是__________。

二、选择题

(1) 下列选项中__________是 Ajax.BeginForm()的参数 ajaxOptions 对象的属性。
　A. actionName　　B. ActionResult
　C. Url　　D. JsonResult

(2) 下列选项中__________是 XMLHttpRequest 对象的属性。
　A. readyState　　B. Url
　C. ajaxOptions　　D. getJSON

(3) 下列选项中__________是 Ajax 请求开始时触发的事件。
　A. ajaxError(callback)　　B. ajaxSend(callback)
　C. ajaxStart(callback)　　D. ajaxStop(callback)

(4) 下列选项中__________是 Ajax 请求结束时触发的事件。
　A. ajaxError(callback)　　B. ajaxSend(callback)

C. ajaxStart(callback)　　　　D. ajaxStop(callback)

(5) 下列选项中__________是 Ajax 请求出现错误时触发的事件。

A. ajaxError(callback)　　　　B. ajaxSend(callback)

C. ajaxStart(callback)　　　　D. ajaxStop(callback)

三、上机练习

上机练习：异步请求学生成绩信息。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 Ajax_mvc。

(2) 在控制器中创建一个返回数据类型为 JSON 的动作方法，方法名自定义。用 Ajax 实现异步请求学生成绩信息的功能，提供一个学生成绩信息类 stuinfo，其代码如下：

```
public class stuinfo
{
    public int Id { get; set; }                 //学生编号
    public string Name { get; set; }            //学生姓名
    public int Math score { get; set; }         //数学成绩
    public int Chinese score { get; set; }      //语文成绩
    public int English_score { get; set; }      //英语成绩
}
```

运行效果如图 12-9 所示。



图 12-9　异步请求学生成绩信息



第 13 章　单 元 测 试

内容摘要

一般来说，我们在编写代码时，一定会反复调试以保证它能够编译通过。如果编译没有通过，没有人会当成工作成果交付给自己的领导。最后代码通过了编译，只能说明它的语法正确，我们却无法保证它的功能实现也完全正确，所以没有人会保证这段代码一定没有 Bug。

要对我们编写的代码进行功能测试，有一个方法就是把它加入项目，进行整体测试。但是，这几乎是不可能的，一些大的项目组中很多人负责的都是一个不完整的模块，而每个人的开发进度不一样，所以实时进行整体测试是不现实的。

那么，我们就要想办法对每一个小单元，甚至是一个小的任务点、一个类或一个方法进行测试。这种测试，就是单元测试。

编写单元测试可以用来验证这段代码的行为是否与我们期望的一致。有了单元测试，可以轻松地检查代码逻辑的正确性。好的单元测试会对应用程序的功能提供可靠的保证。

本章首先来了解一下单元测试的意义，以及什么是测试驱动开发(TDD)，最后将 TDD 思想运用到我们的 MVC 项目中，分别对路由、控制器进行测试。

学习目标

- 了解单元测试的意义
- 了解测试驱动开发的思想
- 掌握 Routing 的测试方法
- 掌握 Controller 的测试方法

13.1 理解单元测试

视频教学：光盘/videos/13/13.1 单元测试 长度：13 分钟

13.1.1 单元测试的意义

一直说单元测试，首先我们要弄懂什么是单元测试。

1．单元测试的概念

单元测试是针对代码单元的独立测试，也叫横块测试。

在.NET Framework 中，程序的最小单元是类。但是从实用角度讲，我们处理程序逻辑或业务逻辑使用的是程序里的方法。如果说以类为单元进行测试则比较复杂，也几乎没有必要，所以单元测试里的代码单元一般指的是类里的方法。

独立的测试是指将代码单元从原始项目中隔离出来，进行单独的测试。

也就是说单元测试是指编写专门的代码单独调用需要测试的方法，进行方法功能验证的工作。

2．为什么要做单元测试

在多个人协同开发的软件项目中，每个人负责一部分代码的编写。多个人负责的各个部分功能可能关联十分紧密，单独运行调试基本上是不可能的事情。

但是，我们又不能百分之百保证我们编写的代码没有任何 Bug，没有一点缺陷，完全可以准确、高效地运行，所以我们需要对各自编写的代码进行一些功能验证。

于是，单元测试诞生了。

3．单元测试的作用

单元测试能够验证代码的行为是否与我们最初期望的一致，它可以最小粒度地对我们的代码进行测试。

单元测试不仅仅是作为无错编码的辅助手段在一次性的开发过程中使用，而且单元测试必须是可重复使用的。无论是在代码编写、软件修改或者移植到新的运行环境的过程中，单元测试都需要正常运行。因此，所有的单元测试都必须在整个软件系统的生命周期中进行维护、使用。

4．什么时候使用单元测试

对于这个问题，我们需要首先了解一下软件开发的流程。

软件开发流程基本上分为分析需求、架构设计、编码、测试以及验收等阶段，单元测试在编码工作以后测试工作的初期执行。业界流行的一个软件开发 V 型流程图详细标明了单元测试的执行时机，如图 13-1 所示。

其实，图 13-1 标示的单元测试可能会给我们造成误解。根据图中标示的顺序，我们很可能以为单元测试将在编码工作完全结束的时候进行，其实不然。

一般来说，单元测试是在每一个程序单元编写完成以后都要编写对应的测试代码进行测

试，然后编写下一个代码单元并进行测试，所以编码工作和单元测试工作在整个工作流程上基本上是同时执行的，只是在逻辑上与编码工作相分离。

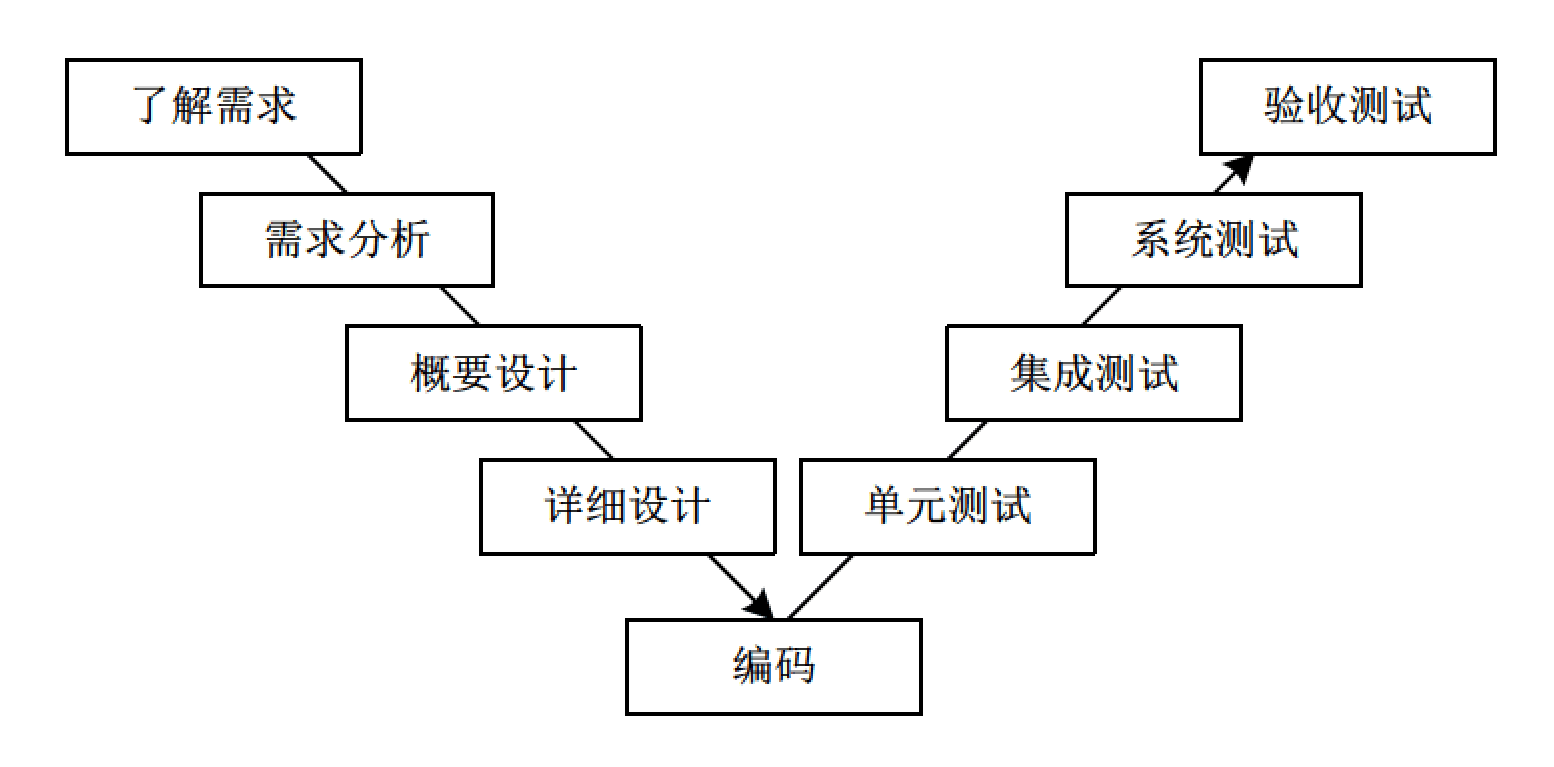


图 13-1　软件开发 V 型流程图

5. 应该由谁来进行单元测试

在协同程序开发中，每个人负责多个程序单元的编写。每一个程序单元的功能应该是负责该单元编码工作的开发人员最为了解，所以编写合适的测试代码验证程序单元的功能完整性应该由每个开发人员负责。

13.1.2　TDD 简介

TDD 的全称是 Test Driven Development，即测试驱动开发。

测试驱动开发是敏捷开发中的一项核心实践和技术，也是一种设计方法论。测试驱动开发的核心思想是在编写程序功能代码(程序单元)之前，先编写测试用例的代码，以便使用测试用例来帮助我们规划程序功能代码设计的性状。

或许上面的解释不容易理解，我们换个角度理解这个概念。从该词组的字面意思理解：测试驱动开发，这里的“驱动”是一个动词，有“使……动”的意思，所以整个词组的意思就是使用测试推动开发进程。



之所以从字面意思理解，是因为有些人可能顺口将“测试驱动”作为一个名词去读，这样将会显得很抽象、很难理解，所以从字面意思上解读更容易体现该词组的意思。

既然 TDD 需要先编写测试用例的代码，再实现该程序单元的功能代码，因此 TDD 的实现基本上可以包括以下内容：

- 构建程序单元的结构。
- 编写单元测试的代码。
- 实现程序单元的功能。
- 测试并在必要时重构程序单元的代码。

下面我们来简单看一下 TDD 各个内容的操作。

1. 构建程序单元的结构

编写测试代码需要引用程序单元。TDD 的第一步，我们需要编写一个程序单元的结构，也就是类和没有实现的程序单元(方法)，代码如下：

```
public class Min
{
    public int GetMin(int[] arrs)
    {
        throw new NotImplementedException();
    }
}
```

这段代码声明了一个获得整型数组中最小值的程序单元结构，不过这里并没有实现它。

2. 编写单元测试的代码

有了程序单元的结构，我们就可以编写单元测试的代码来完善一个单元测试项了。在另一个单元测试类中添加如下代码：

```
[TestMethod]
public void TestMin()
{
    //初始化一个整型数组
    int[] arrs = { 21, 5, 23, 74, 25, 1, -3, 345 };
    //初始化一个求最小整数的对象
    Min min = new Min();
    //调用 GetMin()方法传入数组，获得最小值
    int result = min.GetMin(arrs);

    //验证测试
    Assert.AreEqual(-3, result);
}
```

上面代码初始化一个整型数组，然后创建一个 Min 对象，并调用 GetMin()方法获得最小值，再使用 Assert(断言)下的 AreEqual()方法来测试所求得的结果与我们设想的是否一致。Assert 会自动返回单元测试的结果。

3. 实现程序单元的功能

从上面的单元测试代码我们可以清楚地看到，我们的 GetMin()方法要返回一个数组中的最小值，所以我们可以在 GetMin()方法中编历传入的数组，取出最小值并返回，具体代码如下：

```
public int GetMin(int[] arrs)
{
    int result = arrs[0];
    foreach (int i in arrs)
    {
        if (result > i)
        {
            result = i;
        }
    }
    return result;
}
```

上面代码遍历数组并进行判断，执行完遍历操作以后 result 的结果就是数组中的最小值，然后使用 return 语句将其返回。

4．测试并在必要时重构程序单元的代码

运行前面创建的测试代码，测试顺利执行通过，如图 13-2 所示。

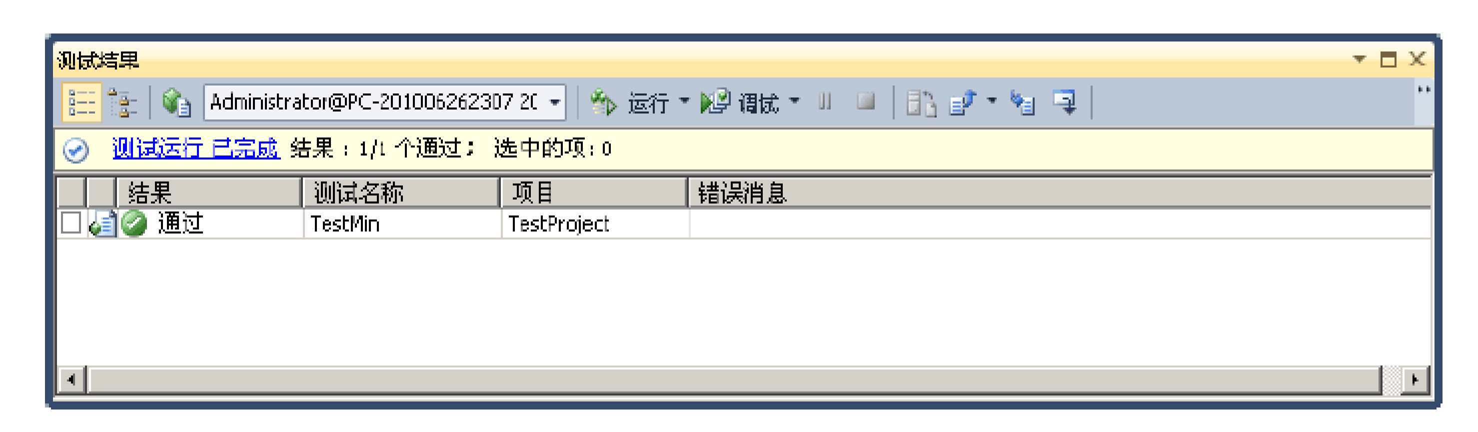


图 13-2　测试执行通过

不过，假如我们在测试代码中传入 GetMin()方法的数组是一个空数组或 null，代码如下：

```
…
//初始化一个整型数组
int[] arrs = { };
…
```

这时候程序肯定会报异常，如图 13-3 所示。

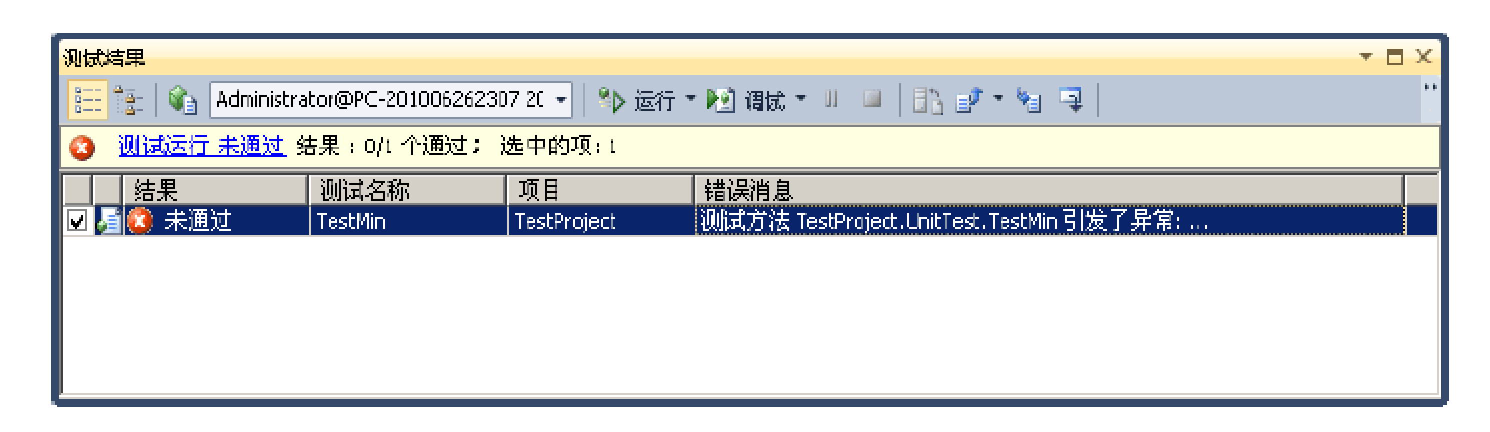


图 13-3　测试未通过

双击错误信息，打开测试结果窗口，如图 13-4 所示。



图 13-4　测试结果

为了提高程序的健壮性，我们修改 GetMin()方法，进行一些数据合法性验证，代码如下：

```
public int GetMin(int[] arrs)
{
    if (arrs == null || arrs.Length == 0)
    {
        throw new ArgumentException("集合不能为空！");
```

```
    }

    int result = arrs[0];
    foreach (int i in arrs)
    {
        if (result > i)
        {
            result = i;
        }
    }
    return result;
}
```

上面代码在检查到传入的数组为 null 或长度为 0 时，主动抛出一个异常信息，拒绝执行下面的操作。

同时我们在测试方法 TestMin() 上面添加一行代码：[ExpectedException(typeof(ArgumentException))]，标记该次测试期望得到一个 ArgumentException 类型的异常。代码如下：

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void TestMin()
{
...
```

如此一改，求整型数组中最小值的程序单元代码相对来说就比较完善了，再使用单元测试代码进行测试，同样可以正常通过。

13.2 使用单元测试验证站点路由

路由选择是 ASP.NET MVC 中一个非常重要而且必不可少的功能。路由配置是整个应用程序的咽喉，路由配置的正确与否直接关系到应用程序能否正确运行。

适当地对路由进行测试，也是整个程序开发过程中不可或缺的一项任务。

 **视频教学：**光盘/videos/13/13.2 使用单元测试验证站点路由 长度：11 分钟

13.2.1 基础知识

从表面上看，路由映射是一组配置信息，用于配置 URL 到对应 Controller 中相应 Action 上的映射关系，实现一组配置功能。但是，这组路由配置信息是直接编辑到 Global.asax 文件中的，而 Global.asax 文件是一个 C#类文件(在使用 C#语言的 ASP.NET MVC 项目中)，在程序发布以后会直接编译到 DLL 类库中。就像一些静态数据项，而不是传统意义上的配置信息。

不过，当考虑到路由配置信息要将请求映射到一个控制器类中的某个方法上时，就会意识到将路由配置信息当做代码来处理也是一个很不错的主意。毕竟路由映射规则应该是程序设计过程中已经确定了的，基本上没有人会在程序发布后修改这些配置信息，所以直接将其设置在代码中，就直接避免了可能出现的意外而使应用程序无法运行的情况。

测试路由的一般模式是将所有的路由添加到 RouteCollection 的本地实例中，然后伪造一个 HTTP 请求进行验证，并根据从请求中解析的数据来判断请求是否被正确解析。

要伪造一个请求，可以自己编写一个测试要用到的 HttpContextBase 类实例，另外还需要一个 HttpRequestBase 类实例。不过，这两个类有很多成员，所以手动编写它们将会是一件非常麻烦的事情。

值得庆幸的是，我们可以使用一个名为 MoQ 的模拟框架，动态地伪造一个类。

技术文档	什么是 MoQ？
MoQ 读做 Mock-you 或者 Mock。它是一个可以在.NET 开发中使用的模拟框架。 它可以作为类型安全和重构友好的模拟库来使用，并且支持模拟接口和类，而且不需要先学习任何其他知识，也不需要专门的经验。 MoQ 框架大量使用了.NET Framework 3.5 中的 Linq 表达式树和 C# 3.0 的 limbda 表达式。 MoQ 可以从 http://code.google.com/p/moq 处下载，下载以后我们会得到一个名为 Moq.dll 的类库文件，在项目中引用它即可使用 MoQ 框架。	

MoQ 的使用非常简单。首先创建一个 Mock 类的对象，同时指定要模拟的类。格式如下：

```
var httpContextMock = new Mock<HttpContextBase>();
```

之后可以使用该对象的 Setup 方法来得到一个模拟项的设置选项，然后就可以调用 Returns() 方法来设置这个项的值了，代码如下：

```
httpContextMock.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath)
        .Returns("~/Prod/Create");
```

最后使用该 Mock 对象的 Object 属性，就可以得到模拟过后的对象，代码如下：

```
httpContextMock.Object
```

13.2.2　实例描述

ASP.NET MVC 允许我们根据实际情况随意制定路由规则。虽然默认有一个通用的路由规则，但是我们经常会根据自己的喜好自定义一些路由规则。

例如在程序中为了起一个有意义的名字，通常会将类名写成全称，但是我不喜欢太长的 URL，所以将 URL 中的 Controller 进行简写，只取前 4 个字母。例如 Product，我将其简写成 Prod。

定义完理论上应该能用了，但为了保险起见，这里我使用单元测试，对其进行验证。

13.2.3　实例应用

【例 13-1】使用单元测试验证站点路由。

(1) 新建一个空的解决方案，命名为 UnitTest。

(2) 添加一个“ASP.NET MVC 2 空 Web 应用程序”，命名为 MvcTest。

(3) 添加一个测试项目，命名为 TestProject。



如果你创建的是“ASP.NET MVC 2 Web 应用程序”，则会提示你是否创建测试项目，如果你创建了，这一步可以省略。

(4) 添加一个单元测试类，命名为 UnitTest。

(5) 在 TestProject 项目中添加对 MoQ 框架的引用，并在 UnitTest 类中添加对 MoQ 命名空间的引用，代码如下：

```
using Moq;
```

(6) 在 UnitTest 类中添加一个测试方法，命名为 TestRouting。

(7) 修改 TestRouting()方法，添加代码如下：

```
[TestMethod]
public void TestRouting()
{
    //创建一个空的路由集合
    RouteCollection rc = new RouteCollection();
    //将路由配置注册到该路由集合中
    MvcApplication.RegisterRoutes(rc);

    //创建一个 Mock 对象
    var httpContextMock = new Mock<HttpContextBase>();

    //设置虚拟请求的相对路径
    httpContextMock.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath)
        .Returns("~/Prod/Create");

    //获得路由集合中与模拟数据匹配的路由信息
    RouteData routeData = rc.GetRouteData(httpContextMock.Object);

    //验证是否有匹配的路由信息
    Assert.IsNotNull(routeData,"没有匹配的路由！");
    //验证指定的 URL 对应的 Controller
    Assert.AreEqual("Product", routeData.Values["controller"]);
    //验证指定的 URL 对应的 Action
    Assert.AreEqual("Create", routeData.Values["action"]);
    //验证指定的 URL 对应的 id 参数
    Assert.AreEqual(UrlParameter.Optional,routeData.Values["id"]);
}
```

上面代码先获得所有的路由规则，然后使用 Mock 对象虚拟了一个 HttpContextBase 对象。模拟请求~/Prod/Create 路径，然后使用 RouteCollection 对象的 GetRouteData()方法来获得路由信息，最后使用 Assert 下的方法进行验证。

(8) 打开 MvcTest 项目中的 Global.asax 文件，添加一条路由配置信息，代码如下：

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    // 参数默认值
    routes.MapRoute(
        "Prod",
        "Prod/{action}/{id}",
```

```
        new { controller = "Product", action = "Index", id =
UrlParameter.Optional }
    );

    // 参数默认值
    routes.MapRoute(
        "Default", // 路由名称
        "{controller}/{action}/{id}", // 带有参数的 URL
        new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

这里在默认的路由规则前面添加了一条路由规则，将虚拟路径为 Prod/{action}/{id}的 URL 映射到 Product 控制器中的对应 Action 上。

13.2.4　运行结果

在 UnitTest 类的代码视图中右击，然后从弹出的快捷菜单中选择【运行测试】命令，打开【测试结果】窗口。稍等片刻，就会执行成功，如图 13-5 所示。

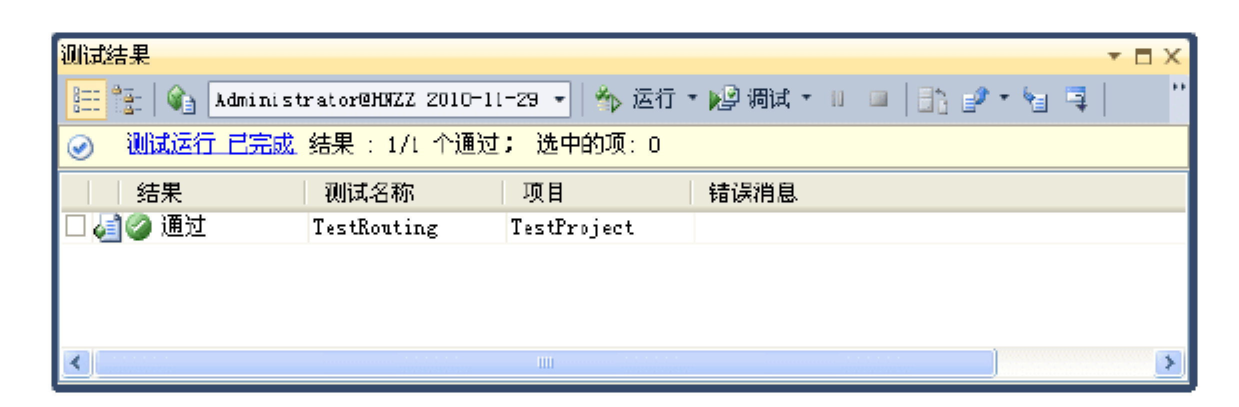


图 13-5　测试成功

当然，想看执行失败很简单。我们修改测试代码中虚拟请求的代码，修改成其他请求地址，例如~/Prod1/Create，主要代码如下：

```
...
    //设置虚拟请求的相对路径
    httpContextMock.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath)
        .Returns("~/Prod1/Create");
...
```

这样，虚拟请求将匹配 Default 路由规则，当然这次请求也就执行失败了。再次测试，在【测试结果】窗口中将显示一个红色图标，如图 13-6 所示。

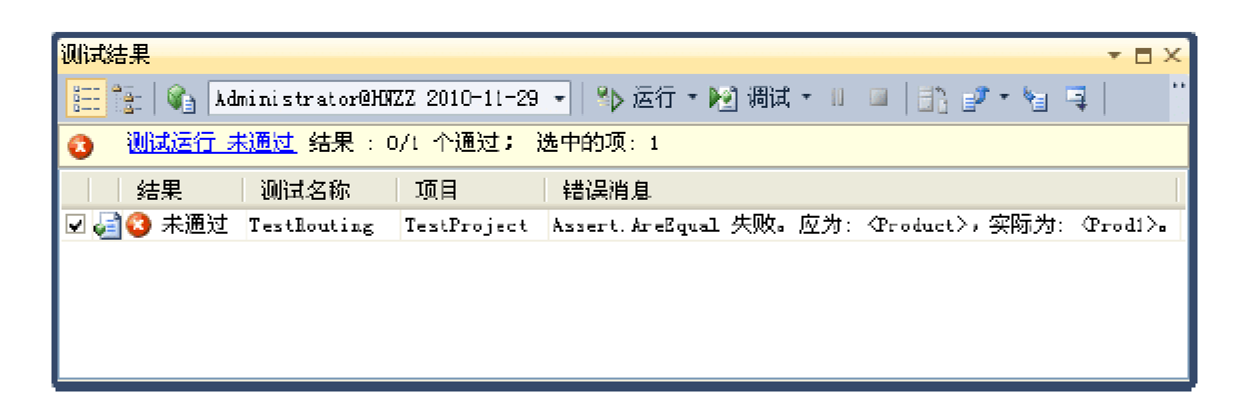


图 13-6　测试未通过

我们看到测试的 Controller 要求为 Product，实际却是 Prod1，很显然使用的是第二个路由规则。

13.2.5 实例分析



源码解析

本实例使用单元测试对路由规则进行测试。

首先我们创建一个空的路由集合，然后使用 MVC 项目中 MvcApplication 类的 RegisterRoutes()方法注册得到所有的路由规则，再使用 Mock 类创建一个执行虚拟请求的对象，并设置其请求的虚拟路径，最后得到与虚拟请求匹配的路由信息，同时取出路由的详细信息，再使用 Assert 进行相应的验证。

13.3 测试 HomeController 的登录功能

在 ASP.NET MVC 中，控制器包含了大量的应用程序逻辑，而且它直接控制着应用程序的响应。整个应用程序的用户界面层的核心就是控制器，控制器健壮与否将直接影响应用程序给用户的感受，所以对控制器进行集中测试意义重大。



视频教学：光盘/videos/13/13.3 测试 HomeController 的登录功能 长度：10 分钟

13.3.1 基础知识

对控制器做测试，其实就是对控制器方法做测试，也就是对 Action 进行测试。

在 ASP.NET MVC 中，正常情况下每个 Action 都会返回一个 ActionResult。我们只要根据每个 Action 的返回值来判断是不是理想的结果即可。

其实这里的 ActionResult 只是一个基类，其中并不包含 ViewData 之类属性的声明，实际上我们在 Action 中使用 View()方法返回的是一个 ViewResult 对象，所以我们要在实现的时候进行确切的类型转换。

在数据验证的时候，我们可以判断返回值是不是特定的 ViewResult 类型的对象，或者判断 ViewResult 中返回的各个数据是否正确，例如 ViewData 中的数据、TempData 中的数据或者 ViewName 是否正确。



一般来说，ActionResult 是一个 ActionResult 对象，不过因为一些特殊需求，可能返回的是 void，JSON 或者其他的对象。这里以 ActionResult 为例，讲解 Controller 测试。

虽然验证是一件简单的事情，但是如何模拟一个请求环境却不是一件容易的事情。如果使用的是一个不需要请求数据的 Action，那么我们可以直接创建一个对应 Controller 的实例，并将 Action 像普通方法一样使用即可。但是如果这个 Action 需要请求数据，例如需要以 Form 方式提交一些数据，那么我们就需要使用一些特殊手段来模拟一些环境了。

首先，我们的 Controller 需要一个上下文 ControllerContext 对象来模拟 Controller 运行的环境。我们可以通过 Controller 对象的 ControllerContext 属性设置给它。

初始化 ControllerContext 对象，需要 RequestContext 和 Controller 对象。构造方法如下：

```
ControllerContext(RequestContext requestContext, ControllerBase controller)
```

ControllerBase 对象就是使用该 ControllerContext 的控制器对象，而 RequestContext 就得自己初始化一个。初始化 RequestContext 对象时需要 HttpContextBase 和 RouteData 对象。

```
RouteContext(HttpContextBase httpContext, RouteData routeData)
```

RouteData 在这里用不到，仅使用无参构造方法创建一个空的对象即可。但是 HttpContextBase 类的对象就需要虚拟一个。

因为 HttpContextBase 对象里面包含了许多从客户端请求的数据(例如 Form)，所以为了方便，我们需要使用 Mock 对象来模拟一个 HttpContextBase 对象。当然，还要虚拟一个 HttpRequestBase 对象，用来封装请求数据。Form 中的数据是一个键/值对集合，所以我们可以使用 NameValueCollection 对象来存放这些数据。

Mock 对象的使用方法非常简单，在上一节我们已经使用过它。例如我们要模拟一个 HttpContextBase 对象，并设置其 Request 属性的值，我们可以使用下面的代码来完成。

```
Mock<HttpContextBase> hcb = new Mock<HttpContextBase>();
hcb.Setup(c => c.Request).Returns(httpRequestValue);
```

上面代码中的 httpRequestValue 是一个 HttpRequestBase 类型的对象。

基本上需要注意的也就这么多了，下面我们来看一个测试 Controller 的实例。

13.3.2　实例描述

我们的 ASP.NET MVC 站点中有一个登录页面，可以接收使用 Form 方式提交过来的用户名和密码，然后将用户名和密码封装到 UserInfo 对象中，作为强类型的 Model 发送到 View 中，并且向 ViewData 中保存两个字符串值：一个 Title 作为页面标题，一个 Message 作为页面输出信息。最后调用 LoginOver 视图来处理用户请求。

13.3.3　实例应用

【例 13-2】测试登录功能的 Controller。

(1) 打开上一节我们使用的项目。

(2) 在 MvcTest 项目中的 Controller 目录下添加一个 Controller，命名为 HomeController。

(3) 在 HomeController 中添加一个 Action，命名为 Login。

(4) 在 Models 目录中添加一个类，用于封装用户信息，命名为 UserInfo。代码如下：

```
public class UserInfo
{
   public string Name { get; set; }
   public string Password { get; set; }
}
```

(5) 打开 TestProject 项目下的 UnitTest 类，添加一个测试方法，命名为 TestController。
(6) 修改 TestController 中的代码，如下所示：

```
[TestMethod]
public void TestController()
{
    //设置用户名和密码，准备向表单中封装
    string username = "zhhh";
    string password = "123";

    //模拟一个表单数据集合
    NameValueCollection form = new NameValueCollection();
    form["username"] = username;
    form["password"] = password;

    //模拟一个请求，使用表单集合中的数据
    Mock<HttpRequestBase> hrb = new Mock<HttpRequestBase>();
    hrb.Setup(r => r.Form).Returns(form);

    //模拟一个 HTTP 上下文
    Mock<HttpContextBase> hcb = new Mock<HttpContextBase>();
    hcb.Setup(c => c.Request).Returns(hrb.Object);

    //创建一个 HomeController
    HomeController home = new HomeController();

    //创建一个 Controller 上下文对象
    ControllerContext controllerContext = new ControllerContext(
        new RequestContext(
            hcb.Object,  /* HttpContextBase 对象 */
            new RouteData()  /* 路由信息，本实例用不到，所以建一个空对象 */
            ),
        home  /* 所使用的 */ );

    //设置控制器的上下文
    home.ControllerContext = controllerContext;

    //执行 Login 动作
    ViewResult vr = home.Login() as ViewResult;

    //得到 Model
    UserInfo user = vr.ViewData.Model as UserInfo;

    //验证 Login 动作返回值是否正确
    Assert.IsNotNull(vr);
    //验证用户信息
    Assert.IsNotNull(user);
    //验证用户名
    Assert.AreEqual(username, user.Name);
    //验证密码
    Assert.AreEqual(password, user.Password);
    //验证视图名称
    Assert.AreEqual("LoginOver", vr.ViewName);
    //验证 ViewData 中的 Title 键的值
```

```
    Assert.AreEqual("Welcome", vr.ViewData["Title"]);
    //验证 ViewData 中的 Message 键的值
    Assert.AreEqual("Welcome Login",vr.ViewData["Message"]);
}
```

上面代码使用 NameValueCollection 集合封装了 UserName 和 Password 的值，然后使用 Mock 类分别虚拟了 HttpRequestBase 和 HttpContextBase 对象，初始化 HomeController 对象，并初始化其 ControllerContext 属性，最后调用 HomeController 对象的 Login 方法，得到执行结果，进行用户登录验证。

(7) 完善 MvcTest 项目中的 HomeController 的 Login 动作，添加相应的处理代码如下：

```
public ActionResult Login()
{
    //获取 Form 请求的字段
    string username = Request.Form["username"];
    string password = Request.Form["password"];

    //封装用户对象
    UserInfo user = new UserInfo()
    {
        Name = username,
        Password = password
    };

    //向视图传递信息
    ViewData["Title"] = "Welcome";
    ViewData["Message"] = "Welcome Login";

    return View("LoginOver", user);
}
```

完成所有编码就可以回到 TestProject 中进行测试了。

13.3.4　运行结果

在 UnitTest 类下面的 TestController()方法中右击，然后从弹出的快捷菜单中选择【运行测试】命令，打开【测试结果】窗口。稍等片刻就会执行成功，如图 13-7 所示。

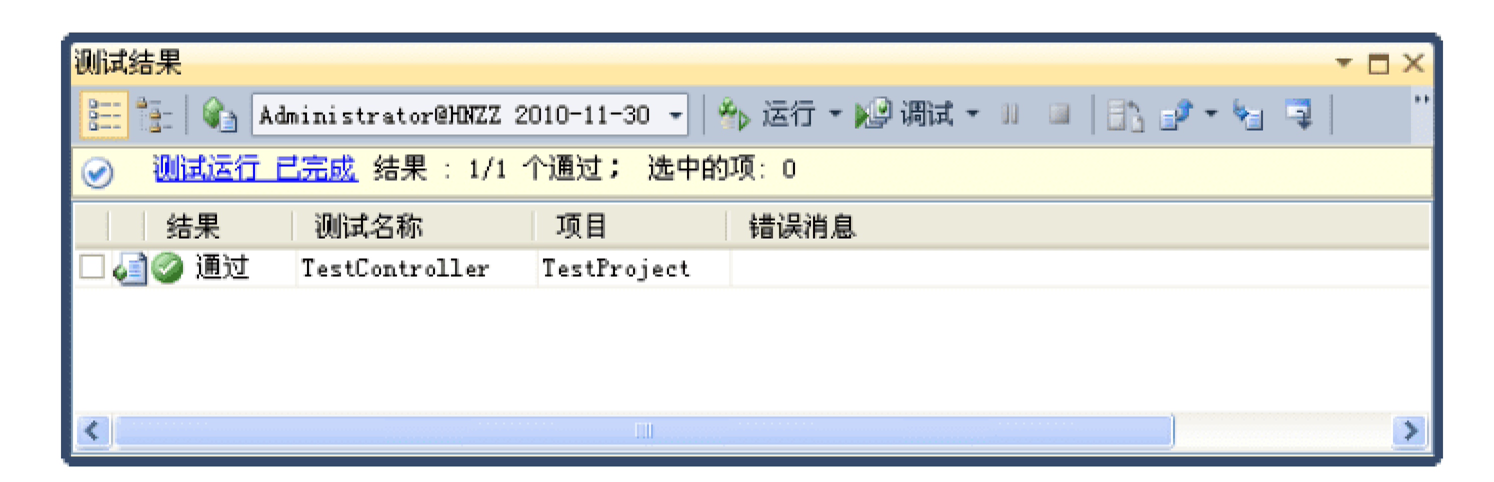


图 13-7　测试成功

13.3.5 实例分析



源码解析

本实例主要是一个测试方法 TestController()。在该方法中先封装一个 Form 表单的键/值对集合，然后使用 Mock 类虚拟 HttpRequestBase 对象并设置其 Form 集合，再使用 Mock 类虚拟 HttpContextBase 对象，并将虚拟的 HttpRequestBase 对象作为请求对象赋给它。创建 HomeController 和 ControllerContext 对象，用来设置 HttpContextBase 对象的 ControllerContext 属性，最后调用 HomeController 对象的 Login 方法获得方法，的返回值，并将其转换成 ViewResult 类型的对象。最后使用 Assert(断言)对 ViewResult 对象中的数据进行验证。

13.4 常见问题解答

13.4.1 TDD 有什么好处



TDD 有什么好处呢？

网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

看到网上有不少人在讲测试驱动开发，感觉太费事了。写一个小功能可能要写一大堆测试代码，它到底有什么好处呢？

【解决办法】TDD 的主旨是高效，可能这个高效并意味着非常高的开发速度。

通常的软件开发过程是先写功能，然后再写测试。甚至有时候只进行某些方面的测试，有省事的则略去了某些细小功能测试，或者忘了对这些模块的测试。

TDD 的思想是以测试推动开发进程。在软件开发之前，每个程序单元的功能都已经确定了。程序员在理解完整个程序需求以后，直接进行开发，有可能因为种种原因考虑不很周全，似乎功能实现没有问题，但其中可能隐藏着非常可怕的 Bug。TDD 促使开发人员先根据程序单元的功能编写测试代码，就像先建一个模型，然后向里面浇注合适功能的代码那样。最后满足所有的测试验证了，才能正常通过测试，这个程序单元才算完成。

这样消除了开发人员主观地对程序单元健壮性的评估，客观地验证每个程序单元的功能以及可能出现的 Bug。

当然，这些操作都需要有大量的代码支持，所以费事在所难免，但是这点“费事”与健壮性非常强的代码相比，有些人还是偏向于使用 TDD。

13.4.2 都说 ASP.NET MVC 提高了可测试性，从哪里体现出来



都说 ASP.NET MVC 提高了可测试性，从哪里体现呢？

网络课堂：http://bbs.itzcn.com/thread-3934-1-1.html

网上很多人在拿 ASP.NET MVC 和 ASP.NET WebForm 进行比较的时候总会说 ASP.NET MVC 提高了可测试性，更容易进行单元测试，保证功能的实现。从哪里体现呢？

【解决办法】首先要肯定一点：ASP.NET MVC 相对于 ASP.NET WebForm 来说确实在很大程度上提高了可测试性。

具体原因如下：

在 ASP.NET WebForm 中能够方便的测试的基本上只有业务类。想要对 Web 窗体进行测试，那是非常困难的。

因为 ASP.NET WebForm 使用了服务器端控件和页面状态，处理起来相对来说非常困难，页面的状态很难进行获取、分析，也很难进行验证和测试。

而 ASP.NET MVC 在可测试性上有很大的进步，因为它将视图层的各个功能进行分开，职责更加明确，分离起来非常方便，环境模拟也非常简单。

例如要测试 Model(Model 就是一些类)，我们只要将其实例化以后就可以进行测试了，为 URLRouting 和 Controller 编写一些模拟数据也很简单。

至于 View，也容易测试。不过对 View 的测试相对来说意义不大，也很少有人大量使用它。

13.5　习　　题

一、填空题

(1) 使用 Mock 对象的__________属性可以得到模拟过后的实体对象。

(2) 假设有一个实体类声明如下：

```
public class User
{
    public string Name{ get; set; }
    …
}
```

请使用 Mock 对象虚拟一个该类的实例，并设置该类的 Name 属性的值为“张浩”，补充以下代码实现该功能:

```
var zh = new Mock<User>();
zh.            (z => z.Name)
    .Returns("张浩");
```

(3) 利用 HttpContextBase 对象的__________属性，可以设置 HttpRequestBase 对象。

二、选择题

(1) 在 ASP.NET MVC 中，单元测试的程序单元是一个__________。

A. 命名空间　　B. 类

C. 方法　　D. 语句

(2) 单元测试应该由__________来完成。

A. 开发人员　　B. 测试人员

C. 架构师　　D. 项目经理

(3) 使用单元测试，我们主要是实现对__________的测试。

A. 程序架构　　　　B. 程序功能

C. 程序模块　　　　D. 程序单元

(4) 在单元测试方法中，使用 Assert(断言)对数据信息进行验证。如果要实现验证一个字符串对象 username 的值是否等于 User 对象的属性 Name 的值，那么下面代码中横线处应该填写的代码是__________。

```
string username = "orbama";
var user = new User() { Name="orbama" };
__________
```

A. `Assert.AreEqual(username, user.Name);`

B. `Assert.IsEqual(username, user.Name);`

C. `Assert.Equals(username, user.Name);`

D. `Assert.AsEqual(username, user.Name);`

三、上机练习

上机练习： 测试产品添加动作。

在 ASP.NET MVC 中，除了 Model，使用单元测试最多的还是 Controller。

本次练习，我们就使用 Visual Studio 内置的测试项目对公司内部信息管理系统的产品管理功能的添加产品动作进行单元测试。

添加产品动作的大致功能如下。

(1) 获得请求中提交的产品信息，封装产品对象。

(2) 调用数据模型，向数据执行添加操作，并获得响应结果。

(3) 根据响应结果向 ViewData 中保存相应的提示信息，例如“添加成功”或“添加失败”。

(4) 在执行 View()方法返回值的时候，向视图发送刚才建立的产品对象，以备在视图中显示出“***产品添加成功”字样。

在单元测试方法中，模拟向 Controller 提交一个存储产品信息的 Form，然后验证控制器动作执行的结果中相应的值。



第 14 章　MVC 博客系统

内容摘要

博客的英文为 Blog，它是一种简单的交流类型网站。通常由简短且经常更新的文章组成，由个人来管理，不定期发表新的文章。

正因为博客沟通的简洁易用性，使它在色彩纷繁的 Internet 世界迅速流行起来，并被誉为是继 E-mail、BBS 和 ICQ 之后出现的第 4 种网络交流方式。

本章将详细讲解创建博客系统的过程，从需求分析入手，以系统功能设计为依据，使用 ASP.NET MVC 2 技术来编写。

学习目标

- 熟悉 ASP.NET MVC 2 项目开发流程
- 了解博客系统所需的各个功能
- 掌握如何设计 MVC 项目的架构
- 掌握 Linq To Sql 在 MVC 中的使用
- 掌握文章模块的实现
- 掌握 MVC 下 jQuery 的应用
- 了解第三方编辑器的使用

14.1 系 统 分 析

做任何事情，准备工作都是必需的，开发应用系统更是如此。在开发博客系统之前，我们要分析该系统的结构，实现哪些功能，并为这些功能分别制作页面，然后考虑该系统需要用到哪些数据表，这些数据表都需要哪些字段以及存储什么数据。

本系统使用的系统开发平台为 Windows XP，Web 服务器为 IIS，数据库服务器为 Microsoft SQL Server 2008，开发工具采用了 Visual Studio 2010，技术架构为 ASP.NET MVC 2。

14.1.1 需求分析

目前，Blog 的内容和用途已经有了很大的变化，有对其他网站的超链接和评论，有关公司、个人、组织机构的新闻到日记、照片、诗歌和散文，甚至科幻小说的发表或转载等。许多 Blog 记录着博友的个人所见、所闻及所想，还有一些 Blog 则是一群人基于某个特定主题或共同利益领域的集体创作。

随着 Blog 的快速发展，它的作用和目的与最初的浏览网页已相去甚远。目前网络上数以千计的博友发表和张贴 Blog 的目的有很大的差异。不过，由于其沟通方式比电子邮件、讨论群组更简单和容易，Blog 已成为家庭、公司、部门和团队之间越来越流行的沟通工具，因此它也逐渐被应用在企业内部网络(Intranet)中。

如果要成为一名博友是一件非常容易的事。只要你愿意，任何人都可以在几分钟之内成为一名博友。

本章的重点不是讲如何成为博友，而是通过所学的技术创建一个属于自己的博客系统。以上背景知识有助于你对博客系统的分析。

根据前面对博客系统的介绍，大概可以设计出所需的主要模块。

- 系统首页　该页面是每个系统必不可少的，在这里提供了到其他页面的链接，给用户浏览各部分信息提供便利。
- 文章模块　文章模块是博客的核心，包括文章栏目、文章标题、文章内容、归档和标签等。
- 用户模块　用户模块提供用户登录、退出以及修改个人资料等功能。
- 管理模块　完善、简洁的管理界面，使管理员能够全面掌握系统的状态，并作出正确的决策，包括对日志的类别、日志内容以及用户资料维护等。

14.1.2 功能设计

根据前面的需求分析，可以确定系统所需的功能模块。通过对各功能模块的扩展，不难拆分出博客系统的功能。我们知道，用户在博客系统中的主要操作有：通过各种方式(按日期或者标签排序)找到感兴趣的文章并阅读文章的内容，同时登录用户还可以发表新文章，或者添加新的栏目等。

图 14-1 为最终博客系统的功能结构图。

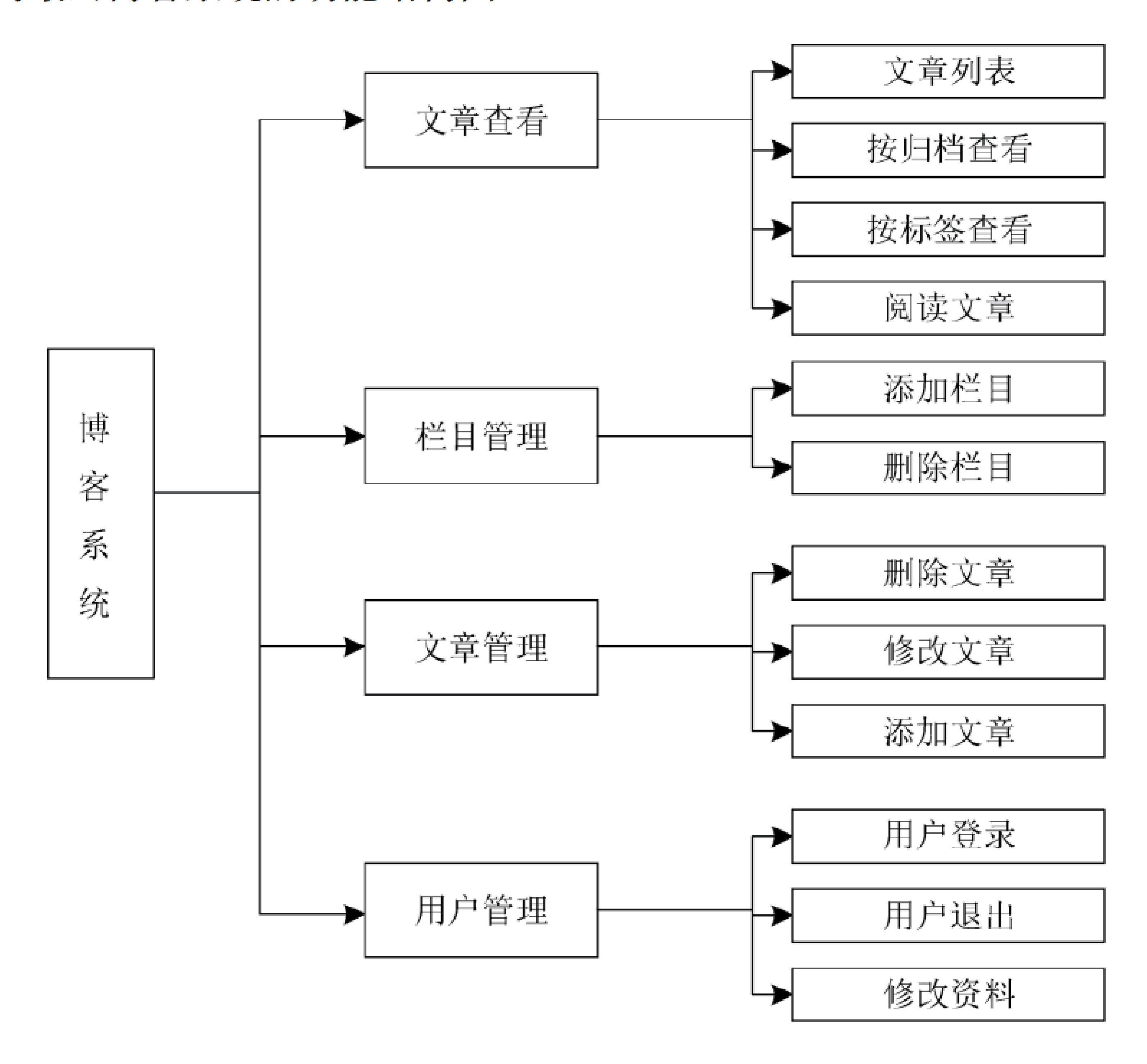


图 14-1　博客系统功能结构图

14.2　数据库设计

本系统采用的是 SQL Server 2008 数据库。打开该数据库管理系统，为本博客系统创建一个数据库，名称为 blog，具体创建过程这里就不再介绍了。

经过上一节的系统结构分析，明确了系统的功能需求，数据库需求设计也变得更加清晰了。在系统中最主要的是对文章信息的存储，例如文章的类别、文章名称以及文章的内容等。

最终确定系统需要使用 5 张表完成。我们在 blog 数据库中建立这些数据表，下面对这些表的名称、描述以及包含字段进行说明。

1．cls 表

cls 表用于保存博客中文章类别信息，例如“学习笔记”、“读书感”等，如表 14-1 所示。

表 14-1　cls 表

字段名	数据类型	是否允许空	备　注
cid	int	否	主键、栏目编号
name	nvarchar(50)	是	栏目名称
demo	nvarchar(50)	是	栏目备注

2．news 表

news 表用于保存具体的文章信息，像文章编号、标题、内容、摘要和发表时间等，如表 14-2 所示。

表 14-2　news 表

字段名	数据类型	是否允许空	备　注
nid	int	否	主键、文章编号
title	nvarchar(50)	是	文章标题
uid	int	是	用户编号
cid	int	是	栏目编号
cont	ntext	是	文章内容
summary	nvarchar(255)	是	文章摘要
tag	nvarchar(50)	是	文章标签
pic	nvarchar(50)	是	图片
ndate	datetime	是	发表时间

3．she 表

she 表用于保存博客系统的全局信息，像博客标题、关键字、具体描述、公司名称和备案信息等，如表 14-3 所示。

表 14-3　she 表

字段名	数据类型	是否允许空	备　注
id	int	否	主键、编号
title	nvarchar(50)	是	标题
keywords	nvarchar(255)	是	关键字
description	nvarchar(255)	是	详细描述
sitename	nvarchar(50)	是	网站名称
company	nvarchar(50)	是	公司名称
tel	nvarchar(12)	是	电话
url	nvarchar(50)	是	网站地址
email	nvarchar(50)	是	邮箱
qq	nvarchar(12)	是	QQ 号码
beian	nvarchar(50)	是	备案信息

4．tag 表

tag 表则用于保存文章中所出现的标签(短语)以及出现次数，如表 14-4 所示。

表 14-4　tag 表

字段名	数据类型	是否允许空	备　注
tid	int	否	主键、标签编号
name	nvarchar(50)	是	标签名称
sum	int	是	出现次数

5．users 表

users 表用于保存博客系统上的用户信息，包括用户名称、密码、级别、注册和登录时间等，如表 14-5 所示。

表 14-5　users 表

字段名	数据类型	是否允许空	备　注
uid	int	否	主键、用户编号
name	nvarchar(50)	是	用户名称
pwd	nvarchar(50)	是	用户密码
jib	int	是	级别
email	nvarchar(50)	是	邮箱
qq	nvarchar(12)	是	QQ 号码
rdage	datetime	是	注册时间
ndate	datetime	是	最后登录时间

14.3　系 统 设 计

前面详细分析了博客系统的功能模块和数据库设计。从本节开始，我们将逐步在 Visual Studio 2010 中编写代码来实现博客系统的各个功能，首先从博客系统项目的创建开始。

14.3.1　创建 MVC 博客项目

在本章开始时就提到，我们要使用 ASP.NET MVC 2 技术来设计博客，而 ASP.NET MVC 2 的最佳开发工具是 Visual Studio 2010，因此我们也使用它作为创建项目的环境。

具体步骤如下。

(1) 启动 Visual Studio 2010，新建一个“ASP.NET MVC 2 空 Web 应用程序”类型的项目，名称定义为 MyMvcBlog。

(2) 打开【解决方案资源管理器】窗口，在当前项目中添加对如下程序集的引用。

- System.Configuration
- System.Xml
- System.Xml.Linq
- System.Data.Linq
- System.Web.Entity

(3) 打开 Web.Config 文件，在 configuration 节点下添加 connectioinStrings 子节点，并配置连接数据库的字符串。

```
    <connectionStrings>
        <add name="blogConnectionString" connectionString="Data
Source=(local);Initial Catalog=blog;Integrated Security=True;User
Instance=false"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
```

14.3.2 创建 Helper

在 MVC 项目的根目录下创建一个名称为 Helpers 的子目录。在这个子目录下保存了我们在博客系统中所用到辅助工具类，主要包括下面 3 个。

- ImageLinkHelper 类　这是一个用于在页面上生成图片链接的类，是一个静态类，包含具有 3 个重载形式的 ImageLink()方法，引用了 System.Web.Mvc 和 System.Web.Routing 命名空间。
- SubmitButtonHelper 类　该类也是一个静态类，可以在页面上生成【提交】按钮，包含一个 SubmitButton()方法，需要引用 System.Web.Mvc 命名空间。
- SqlHelper 类　这是一个数据库连接的抽象类，封装了数据库访问的通用代码，不允许被实例化，在应用时直接调用即可。

以上 3 个类都位于 Helpers 目录下，因此都具有相同的命名空间 MyMvcBlog.Helpers。



这里仅对这 3 个 Helper 类的作用进行了介绍，具体实现代码读者可以参考实例源代码。

14.3.3 创建母版页

为了使博客系统的页面风格保持一致，我们创建了两个母版页来区别前台和后台的风格，这两个母版页均位于 Views 目录下。下面介绍具体的创建过程。

(1) 右击 Views 目录，选择【添加】|【新建项】命令，然后在弹出的【添加新项】对话框中选择【MVC 2 视图母版页】模板，并定义名称为 Main.Master，如图 14-2 所示。

(2) 单击【添加】按钮，创建该母版页，然后对页面进行布局调整。最终需要包含 TitleContent 和 MainContent 两个模板，其中 TitleContent 用于设置内容页的 title 标题，MainContent 用来显示具体的内容。

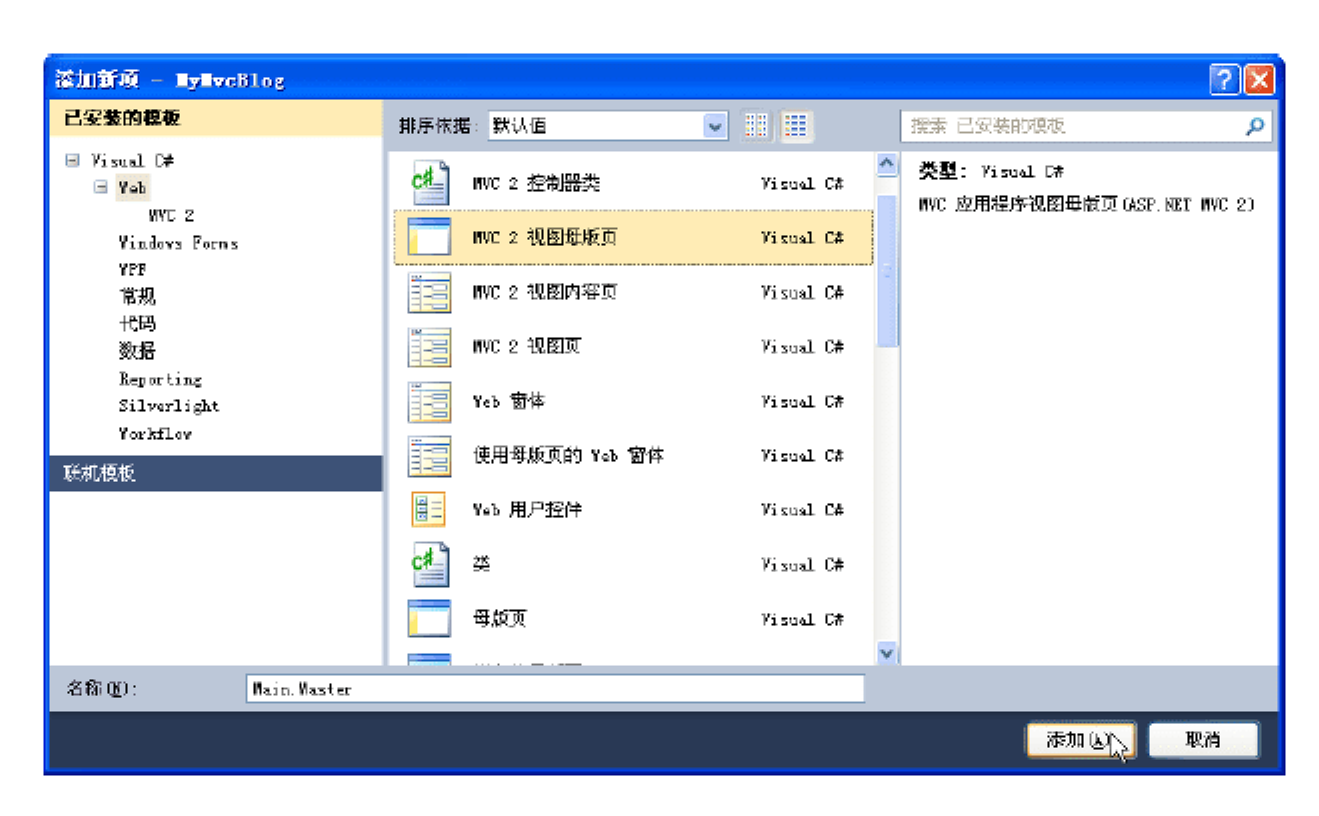


图 14-2　创建 MVC 母版页

(3) 下面对 Main.Master 母版页中的重要代码进行介绍。首先在页面的最底部添加如下两行语句，引入命名空间。

```
<%@ Import Namespace="MyMvcBlog.Models" %>
<%@ Import Namespace="MyMvcBlog.Helpers" %>
```

(4) 在 head 与 body 之间添加如下代码：

```
<%
    DataClasses1DataContext db = new DataClasses1DataContext();
    var b = db.she.First();
    var zuixin = db.news.OrderByDescending(d => d.ndate).Skip(0).Take(10);
    var lanm = db.cls.OrderByDescending(d => d.cid);
    var tags = db.tag.OrderByDescending(d => d.sum).Skip(0).Take(10);
    System.Data.SqlClient.SqlDataReader dr =
SqlHelper.ExecuteReader(SqlHelper.conn, System.Data.CommandType.Text,
"select max(ndate),count(ndate) from news group by year(ndate),month(ndate)
order by  max(ndate) desc");
    List<dang> n = new List<dang>();
    while (dr.Read())
    {
        dang d1 = new dang();
        d1.name = Convert.ToDateTime(dr[0]).ToString("yyyy 年 MM 月");
        d1.sum = Convert.ToInt32(dr[1]);
        d1.id = Convert.ToDateTime(dr[0]).Year.ToString() + "-" +
Convert.ToDateTime(dr[0]).Month.ToString();
        n.Add(d1);
    }
    dr.Dispose();
%>
```

(5) 上面这段代码用来从数据库中读取数据，为后面的显示作准备。在页面的合适位置添加网站的导航条，这里用到了 MVC 中的 HtmlHelper 类，代码如下：

```
  <ul>
     <li><%=Html.ActionLink("网站首页", "index",new{}, new { @class =
"n0"})%></li>
     <li> <%=Html.ActionLink("文章归档", "archive", new { }, new { @id =
"n1" })%></li>
     <li><%=Html.ActionLink("所有标签","tag",new{ },new{ @id="n2" })%></li>
```

```
    <li> <%=Html.ActionLink("内容搜索", "search", new { }, new { @id =
"n3" })%></li>
    <li><%=Html.ActionLink("内容订阅", "rss", new { }, new { @id = "n5" })%></li>
    <li><a href="#">留言</a></li>
</ul>
```

(6) 在右侧添加“文章栏目”模块，用于显示在博客系统中创建的文章类别，代码如下：

```
<h2>文章栏目</h2>
<ul>
    <%foreach (var d in lanm)
      { %>
    <li>
        <%= Html.ActionLink(d.name + "(" + d.news.Count + ")", "index", new
{ id = "c" + d.cid })%>
    </li>
    <%} %>
</ul>
```

(7) 添加“文章归类”模块，该模块将以月为单位显示发表文章的数量以及月份，代码如下：

```
<h2>文章归类</h2>
<ul>
    <%foreach (var d in n)
      { %>
    <li><%= Html.ActionLink(d.name+"("+d.sum+")" , "index",
new{id="d"+d.id})%>
    </li>
    <%} %>
</ul>
```

(8) 添加“最新文章”模块，该模块将显示最近发表的 10 篇文章，代码如下：

```
<h2>最新文章</h2>
<ul>
    <%foreach (var d in zuixin)
      { %>
    <li><%= Html.ActionLink(d.title , "show", new{id=d.nid})%></li>
    <%} %>
</ul>
```

(9) 再添加一个“Tag 标签”模块，该模块会显示所有文章中某个标签(短语)出现的次数，代码如下：

```
<h2>Tag 标签</h2>
<ul>
    <%foreach (var d in tags)
      { %>
    <li>
        <%= Html.ActionLink(d.name+"("+d.sum+")" , "index",
new{id="t"+d.name})%></li>
    <%} %>
</ul>
```

(10) 在母版页的最底部添加如下代码来读取后台设置的版权信息。

```
<div id="footer">
    <p>  Copyright © 2010   <%=b.company %>. All Rights Reserved
        <%=b.beian %><br />
          <a href="<%=b.url %>" target=" blank"><%=b.sitename %>
            版权所有</a>
    </p>
</div>
```

至此，关于前台使用的母版页 Main.Master 就介绍完了。下面创建后台使用的母版页，名称为 admin.Master，另外需要注意在图 14-2 所示的对话框中，应选择【母版页】模板。

与前台的母版页一样，这里创建了 TitleContent 和 MainContent 两个模板项，其中 TitleContent 用于设置内容页的 title 标题，MainContent 用来显示具体的内容。

admin.Master 母版页在后台管理系统中使用，因此相比前台的母版页在导航条上有些区别。更多的是与管理有关的链接，这部分代码如下：

```
<ul>
  <li><a href="/">[1] 博客首页</a></li>
  <li><a href="/admin">[2] 后台首页</a></li>
  <li><a href="/users/login">[3] 登录系统</a></li>
  <li><a href="/cls">[4] 栏目管理</a></li>
  <li><a href="/news">[5] 文章管理</a></li>
  <li><a href="/users">[6] 用户管理</a></li>
  <li><a href="#">[7] 亲朋好友</a></li>
  <li><a href="#">[8] 看看微博</a></li>
  <% if (Request.IsAuthenticated)
    { %>
     <li> <a   href="/users/logout" >[9] 退出系统</a></li>
  <%} %>
</ul>
```

可以看到，如果以非正常模式请求该页面，将不会显示“退出系统”链接。



普通母版页与 MVC 母版页的主要区别就是继承的类不同。前者继承自 System.Web.UI.MasterPage 类，后者则继承自 System.Web.Mvc.ViewMasterPage 类。

14.3.4　创建 Linq To Sql 实体

采用 MVC 架构之后，一个最明显的区别就是不用像传统三层架构那样编写繁琐的数据库访问和 DAL 层。取而代之的是使用 Linq To Sql 来完成它的工作，而且还可以省去 Model 层的编写，简直太激动人心了。

下面我们来看看这个神奇的 Linq To Sql 究竟如何操作。

(1) 右击 Models 目录，选择【添加】|【新建项】命令，然后在弹出的【添加新项】对话框中选择【Linq To SQL 类】模板，并定义名称为 DataClasses1.dbml，如图 14-3 所示。

(2) 单击【添加】按钮，确定创建并进入 Linq To Sql 设计器。从【服务器资源管理器】窗格中新建一个到实例数据库的链接，然后将其中的表依次拖动到设计器内。

(3) 在设计器内右击并选择【添加】|【关联】命令，打开【关联编辑器】对话框。在这里设置将 news 表的 cid 关联到 cls 表的 cid，以及将 news 表的 uid 关联到 users 表的 uid，图 14-4 为编辑时的对话框。

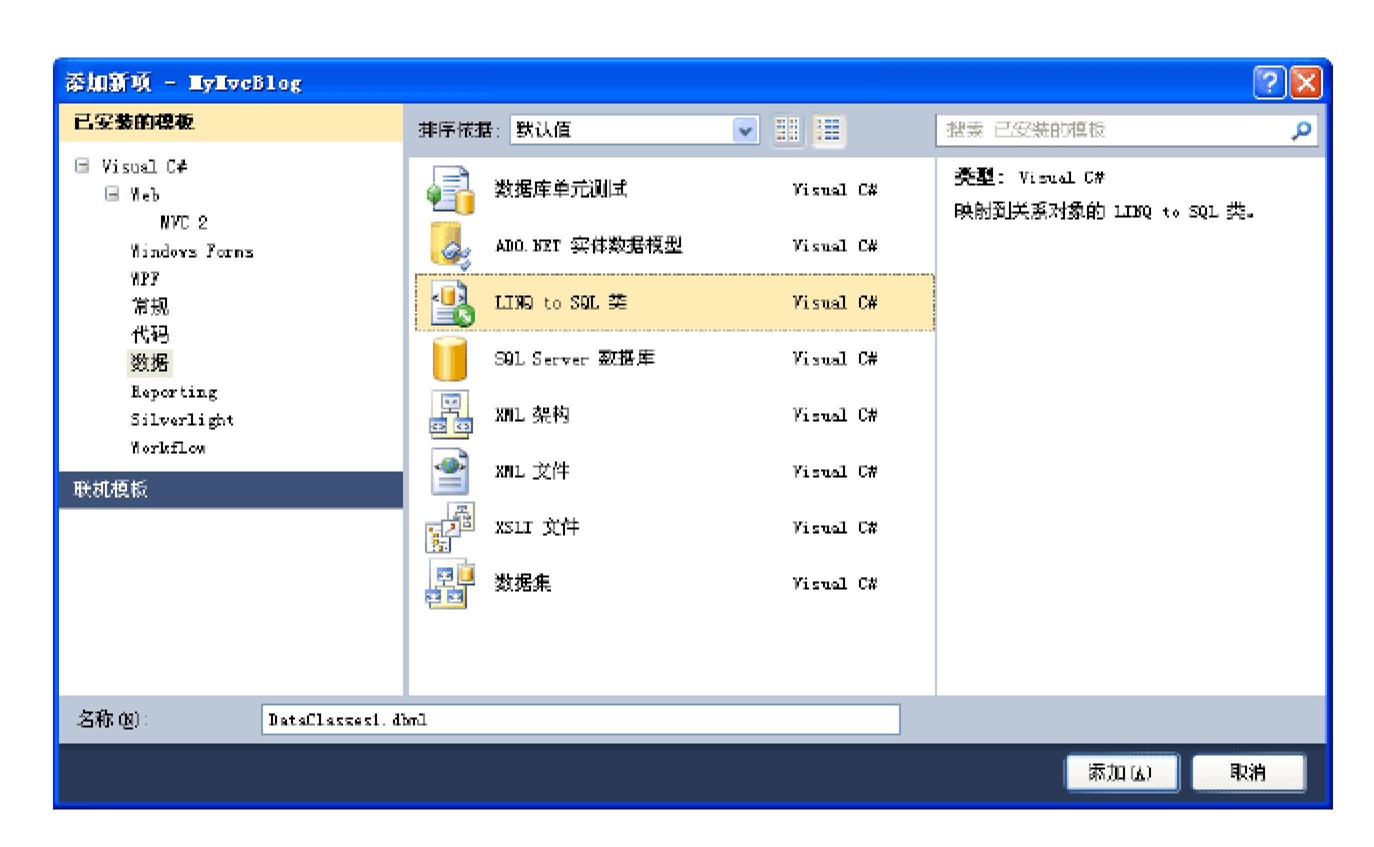
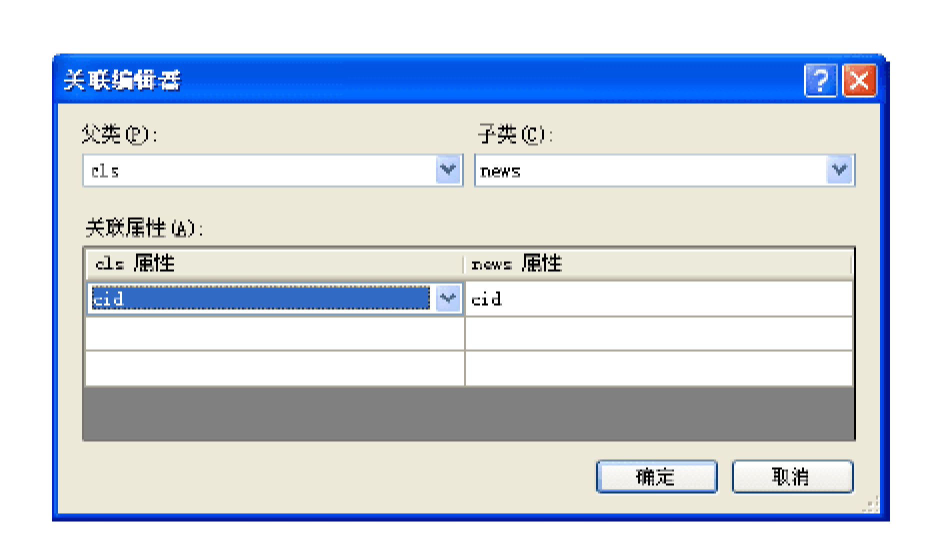


图 14-3　创建 Linq To Sql 类



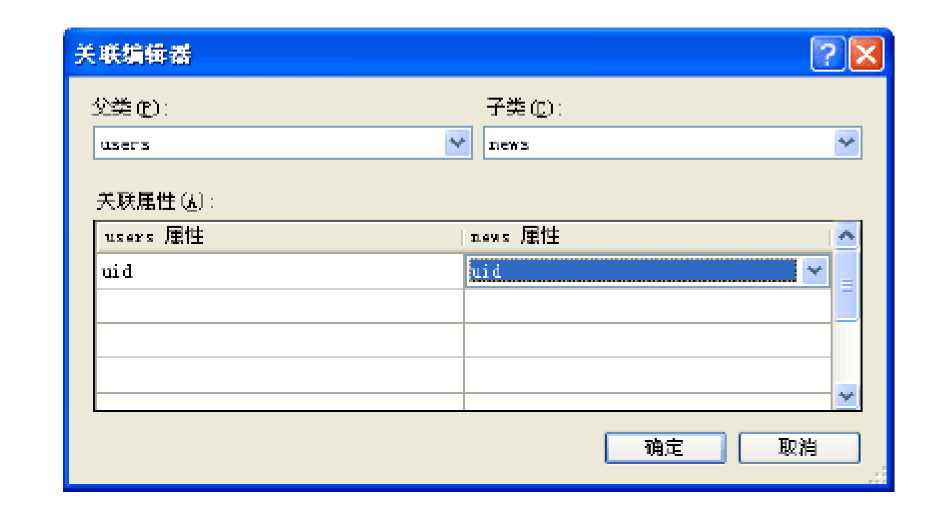


图 14-4　编辑关联

(4) 单击【确定】按钮，完成编辑。此时 cls 表、news 表和 users 表三者之间就有了关联，图 14-5 为最终的类结构图。

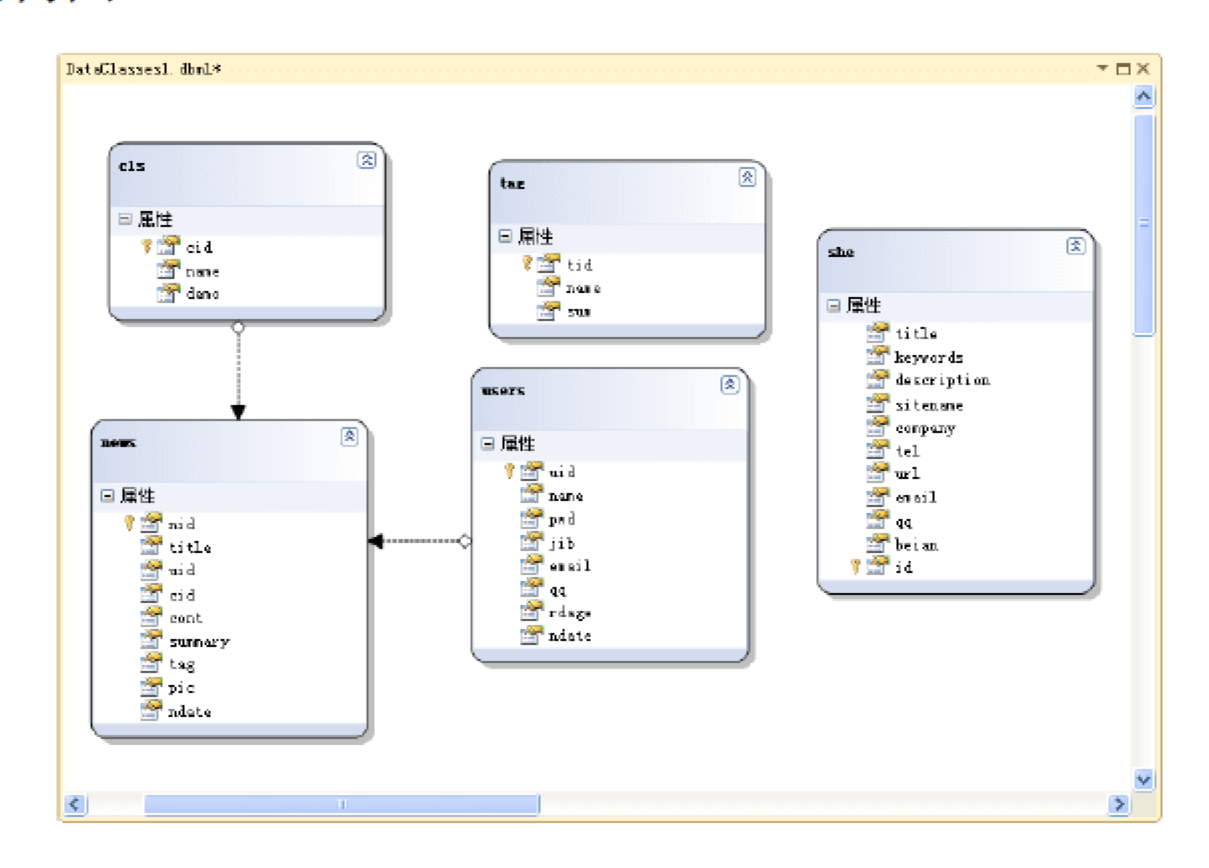


图 14-5　Linq To Sql 类结构图

14.4　文 章 模 块

经过系统设计的几个步骤，系统的基本 MVC 框架就搭建完成了。接下来的工作就是在这个框架上实现博客系统的功能，这里从文章模块开始，因为它是博客系统的核心模块。主要功能有：提供一个带分页的文章列表，查看文章的详细内容，以及按类别、日期或者标签查看文

章等。

在开始文章模块的具体编码之前，我们还需要创建一个名称为 Home 的 Controller。本节的所有 MVC 代码都是基于该 Controller 的，采用的母版页都为 Main.master。

14.4.1 查看文章列表

查看文章列表是我们在运行博客系统之后看到的第一个页面，称为首页。在这里会将所有的文章以每页 5 篇的形式显示出来。对于每一篇文章，将会显示标题、发表时间、摘要、标签和栏目等信息。

当然，也可以通过单击文章的标题查看详细内容。

默认首页实际请求的是 HomeController 的 Index()方法。创建该方法并添加实现代码，部分如下：

```
DataClasses1DataContext db = new DataClasses1DataContext();
public ActionResult Index(string id)
{
    IQueryable<news> n = db.news.OrderByDescending(d => d.ndate);
    if (id.Contains("t"))
    {
        n = from d in db.news
            where (d.tag + " ").Contains(id.Substring(1) + "")
            orderby d.ndate descending
            select d;
    }
    else if (id.Contains("c"))
    {
        n = from d in db.news
            where d.cid == Convert.ToInt32(id.Substring(1))
            orderby d.ndate descending
            select d;
    }
    else if (id.Contains("d"))
    {
        var s = id.Substring(1).Split('-');
        n = from d in db.news
            where Convert.ToDateTime(d.ndate).Month.ToString() == s[1]
            && Convert.ToDateTime(d.ndate).Year.ToString() == s[0]
            orderby d.ndate descending
            select d;
    }
    if (Request["page"] != null)
    {
        ViewData["page"] = SqlHelper.Pager(Request.RawUrl.Substring(0,
Request.RawUrl.IndexOf("?")), Convert.ToInt32(Request["page"]), n.Count(), 5,
3);
        n = n.Skip((Convert.ToInt32(Request["page"]) - 1) * 5).Take(5);
```

```
            }
            else
            {
                ViewData["page"] = SqlHelper.Pager(Request.RawUrl, 1, n.Count(),
5, 3);
                n = n.Skip(0).Take(5);
            }
            return View(n);
        }
```

与上述代码相对应，我们还需要创建一个名为 Index.aspx 的视图文件。在这里利用获取的数据源集合来定义显示的布局，最终代码如下：

```
  <%
        foreach (var d in Model)
        { %>
    <div class="post">
        <h1 class="title">
            <%=Html.ActionLink(d.title, "show", new { id = d.nid })%></h1>
        <p class="meta">
            <a href="#">
                <%=d.users.name %></a>于
            <%=d.ndate %>发表</p>
        <div class="entry">
            <br />
            <%=d.summary %>
            <br />
            <%=Html.ActionLink("详细...", "show", new { id = d.nid }, new { @class
= "xiangxi" })%>
            <p>
                标签:
                <%
                    if (d.tag != null)
                    {
                        var ss = d.tag.Split(' ');
                        foreach (var s in ss)
                        {
                %>
                <%=Html.ActionLink(s,"index",new{id="t"+s} )%>
                <%}
                    }%>
                  栏目:

<%=Html.ActionLink(d.cls.name,"index",new{id="c"+d.cid}) %></p>
        </div>
    </div>
    <%} %>
    <div class="page">
        <%=ViewData["page"] %>
    </div>
```

如果文章数量小于 5 篇，那么保存在 ViewData["page"]中的分页导航将不会显示。图 14-6 为运行后的查看效果。



图 14-6　查看文章列表

14.4.2　查看文章详情

查看文章详情是博客系统中最重要的模块之一，作为博客系统为最终用户提供浏览文章内容的功能是必不可少的。

阅读博客中文章的内容才是浏览者登录博客网站的主要目的，而非仅仅只查看文章的标题和摘要。在图 14-6 中单击文章标题或者“详细”链接就会进入文章的阅读页面。

在 HomeController 中添加名为 Show 的 Action，它带有一个 int 类型参数来表示要查看的文章编号。具体实现代码如下：

```
public ActionResult Show(int id)
{
    var nl = db.news.Where(d => d.nid == id);
    if (nl.Count() > 0)
        return View(nl.First());
    return View();
}
```

为 Show 添加视图文件 Show.aspx，在这里制作显示的布局，包括文章的标题、文章作者、发表时间、文章正文内容、使用的标签以及所属栏目等。

最终布局可以参考如下代码：

```
<div class="post">
    <h1 class="title">
        <%=Html.ActionLink(Model.title,"show",new{id=Model.nid}) %></h1>
    <p class="meta">
        <a href="#">
            <%=Model.users.name%></a>于  <%=Model.ndate%>发表</p>
    <div class="entry">
        <br />
        <%=Model.cont%>
        <br />
        <p>
            标签:
            <%
                if (Model.tag != null)
                {
```

```
                    var ss = Model.tag.Split(' ');
                    foreach (var s in ss)
                    {
             %>
             <%=Html.ActionLink(s,"index",new{id="t"+s} )%>
             <%}
                 }%>
           栏目： <%=Html.ActionLink(Model.cls.name, "index", new
{ id = "c" + Model.cid })%></p>
       </div>
    </div>
    <p>
       <%: Html.ActionLink("返回博客首页", "Index") %>
    </p>
```

上述代码中的数据来自 MyMvcBlog.Models.news 模型类，在视图文件中使用 Model 关键字来引用该类。

在显示文章标签时，使用 foreach 语句遍历 tag 中的值，并通过 Html.ActionLink 组件产生一个动作链接。最后一行显示了一个返回博客系统首页的链接，整个页面运行效果如图 14-7 所示。



图 14-7　查看文章详情

14.4.3　按归档查看

最初的博客系统是以流水形式的日志出现，按归档查看功能提供了以月为单位的日志汇总。它会显示一个列表，清晰地列出某月中发表了多少篇日志。

在 HomeController 中添加名为 Archive 的 Action，具体实现代码如下：

```
  public ActionResult Archive()
  {
     List<dang> n = new List<dang>();              //创建一个保存数据的集合
     SqlDataReader dr = SqlHelper.ExecuteReader(SqlHelper.conn,
System.Data.CommandType.Text, "select max(ndate),count(ndate) from news group
by year(ndate),month(ndate) order by  max(ndate) desc");
     while (dr.Read())
```

```
        {
            dang d1 = new dang();
            d1.name = Convert.ToDateTime(dr[0]).ToString("yyyy 年 MM 月");
            d1.sum = Convert.ToInt32(dr[1]);
            d1.id = Convert.ToDateTime(dr[0]).Year.ToString() + "-" +
Convert.ToDateTime(dr[0]).Month.ToString();
            n.Add(d1);
        }
        return View(n);
    }
```

为 Archive 添加视图文件 Archive.aspx，在这里制作显示的布局，包括添加一个 ul 列表，显示文章发表的月份以及数量等。

最终布局可以参考如下代码：

```
  <div class="post">
      <h1 class="title">所有文章归档</h1>
      <div class="entry">
          <ul>
              <% var n = Model;
                 foreach (var d in n)
                 { %>
             <li>
              <%= Html.ActionLink(d.name + "(" + d.sum + ")", "index", new { id
= "d" + d.id })%>
             </li>
              <%} %></ul>
      </div>
      <p>
          <%: Html.ActionLink("返回博客首页", "Index") %>
      </p>
  </div>
```

在上述代码中，Model 是保存了页面所需的数据，将它赋予变量 n，然后利用 foreach 语句遍历 n，输出一个带动作的链接。整个页面运行效果如图 14-8 所示。

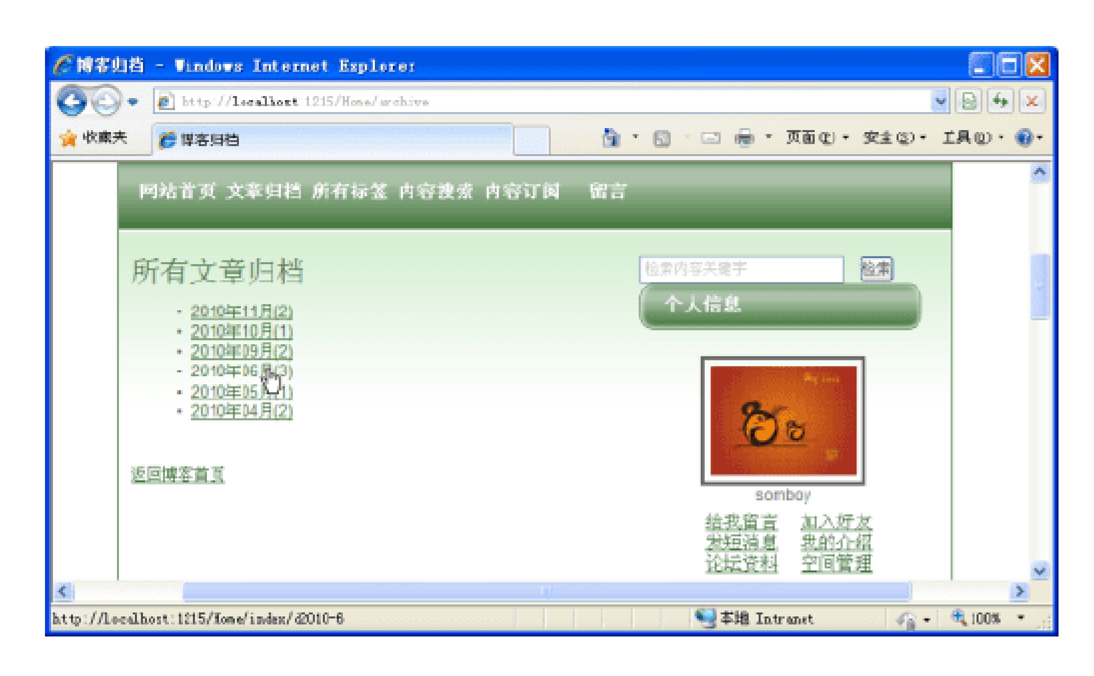


图 14-8　按归档查看

14.4.4　按标签查看

按标签查看与按归档查看的实现方法类似，这里新建一个名为 Tag 的 Action，用于获取标签信息并按降序排列。

具体实现代码如下：

```
public ActionResult Tag()
{
    var t = from d in db.tag
            orderby d.sum descending
            select d;
    return View(t);
}
```

可以看到，由于使用了 Linq To SQL 技术，整个 Model 变得很简单，仅用一行语句即可完成数据库查询功能，对应的数据表为 tag。

创建视图文件 Tag.aspx。根据上节介绍的方法遍历 Model，并依次获取标签名称、标签出现的次数，最终页面运行效果如图 14-9 所示。

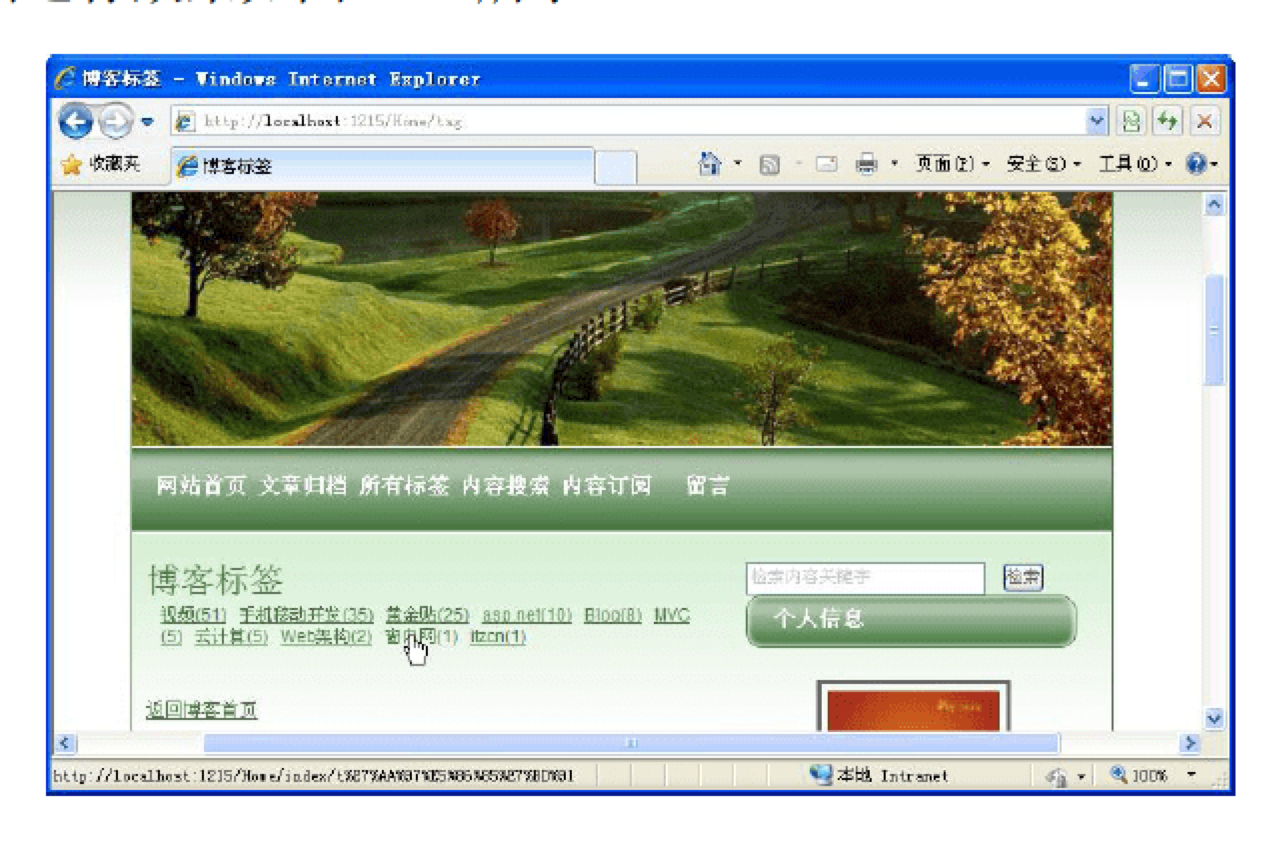


图 14-9　按标签查看

至此，博客系统的前台就全部制作完成了。

14.5　用户管理模块

从本节开始，我们将介绍如何实现用户管理模块。在用户管理模块中，浏览者将成为博客的主体，可以对博客里的文章进行管理，像修改一篇文章或者添加一个栏目等。

用户管理模块使用 admin.Master 作为母版页，Controller 名称为 Users。

14.5.1　用户登录

为了区分普通浏览者与系统用户的特殊身份，在系统中设置了登录入口，只有从登录入口成功登录后才能执行管理操作。

在 UsersController 中新建一个名为 login 的 Action，并返回默认视图。在 login.aspx 下制作用户登录的表单，代码如下所示：

```
<fieldset>
    <legend>用户登录</legend>
```

```
      <%using (Html.BeginForm())
        { %>
      <p> <br />
        用户名：<input type="text" name="name" style="width: 120px" /><br />
        密    码：<input type="password" name="pwd" style="width:
122px" /><br />
          <input type="submit" value="登录" /> <br />
      </p>
      <%} %>
   </fieldset>
```

如上述代码所示，BeginForm()方法将创建一个以 POST 方式提交的表单。提交的地址为本页面，即 users/login。

接下来创建 login()方法，该方法接收 POST 过来的表单数据并验证是否正确。

```
    [AcceptVerbs(HttpVerbs.Post)]                          //仅在 POST 方式时有效
    public ActionResult login(string name, string pwd)
    {
        var d2 = from d in db.users
                 where d.name == name &&
                 d.pwd == SqlHelper.md5(pwd)
                 select d;                                 //执行查询
        if (d2.Count() > 0)                                //如果有记录
        {
            FormsAuthentication.SetAuthCookie(name, true);  //保存当前用户信息
            return RedirectToAction("index", "admin");      //跳转到
admin/index
        }
        return View();
    }
```

这里的[AcceptVerbs(HttpVerbs.Post)]属性使 login()方法仅在 POST 方式提交时才有效，其中包含的 name 参数为用户名，pwd 参数为密码。在执行查询时，如果包含记录，则 Count()>0 条件成立，此时保存当前用户信息并跳转到管理首页。

图 14-10 为用户登录的页面效果。

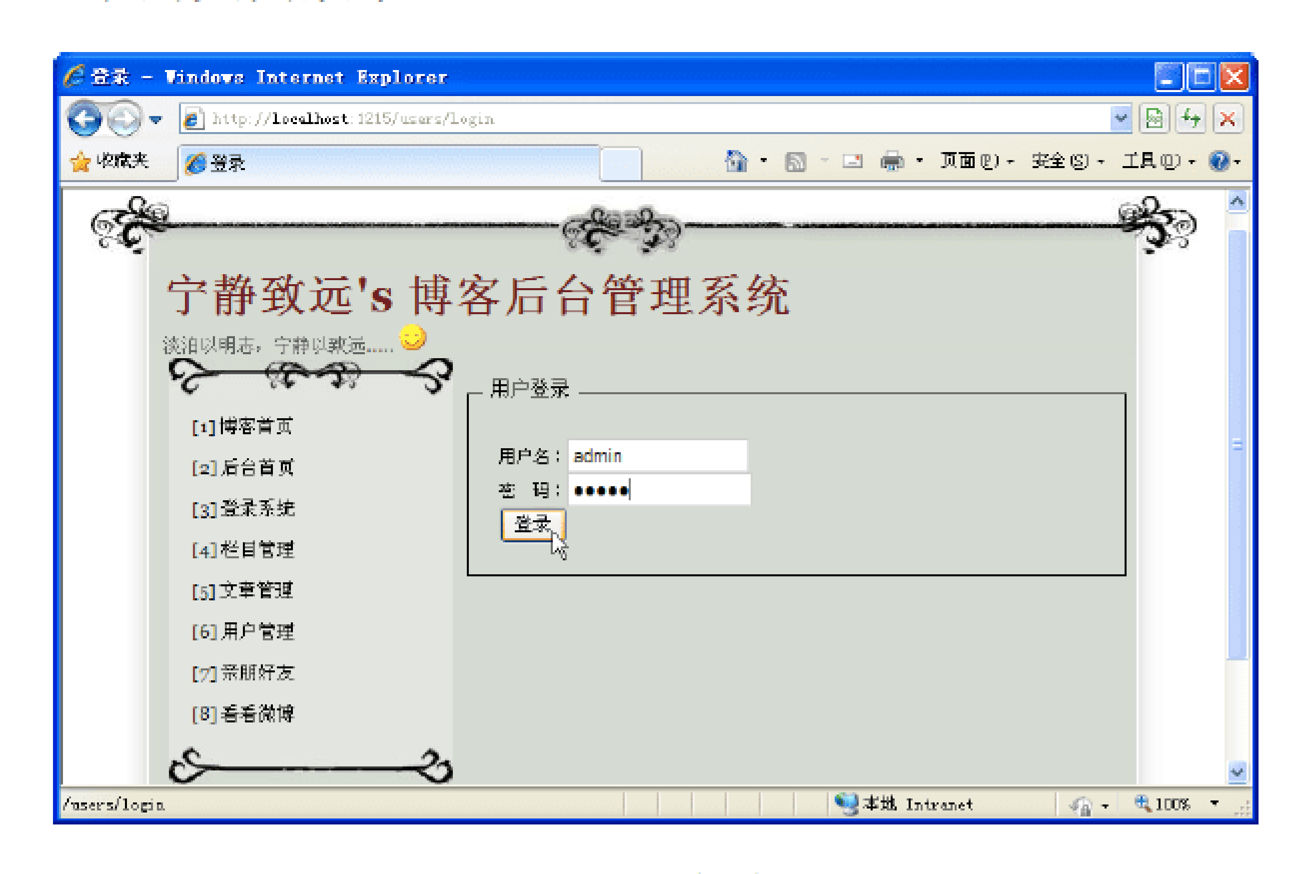


图 14-10 用户登录页面

14.5.2 用户退出

对于一个系统而言，除了要有快捷的操作方式、稳定的性能和美观的界面外，安全因素也是必不可少的。在本实例中用户使用过系统后，需要安全退出系统，防止别人对系统的破坏。

下面介绍的退出代码虽然简单，却是任何系统中不可缺少的部分。

```
public ActionResult logout(string name, string pwd)
{
    FormsAuthentication.SignOut();
    return RedirectToAction("index", "home");
}
```



在本系统中，只有当用户成功登录之后，才可以在管理菜单中看到退出链接，这是靠 Request.IsAuthenticated 的值来实现的。

14.5.3 修改资料

此功能提供了当用户登录后，可对最初的信息进行修改。在这里首先会显示一个用户列表，通过单击“编辑”链接，可以指定要修改的用户。

在 UserController 中新建一个名为 Index 的 Action 并追加[Authorize]属性，代码如下：

```
[Authorize]
public ActionResult Index()
{
    var u = db.users;
    return View(u);
}
```

在上述代码中，u 变量是 Linq To Sql 中 DataClasses1DataContext()类的实例。调用 users 属性可获取 users 数据表中的所有数据，将它传递到视图中。

创建 Index 对应的视图文件 Index.aspx，在这里除了以表格方式罗列用户之外，还提供了一个新建用户的链接。这部分代码如下：

```
<p>
    <%= Html.ActionLink("新建一位用户", "Create") %>
</p>
<fieldset>
    <legend>用户管理</legend>
    <table style="width: 100%">
        <tr>
            <th>编号</th>
            <th>名称</th>
            <th>级别</th>
            <th>E-mail</th>
            <th>QQ</th>
            <th>注册时间</th>
            <th></th>
        </tr>
        <% foreach (var item in Model)
```

```
            { %>
        <tr>
            <td><%= Html.Encode(item.uid) %> </td>
            <td><%= Html.Encode(item.name) %></td>
            <td><%= Html.Encode(item.jib) %> </td>
            <td><%= Html.Encode(item.email) %></td>
            <td><%= Html.Encode(item.qq) %> </td>
            <td><%= Html.Encode(String.Format("{0:g}", item.rdage)) %> </td>
            <td>
                <%= Html.ActionLink("编辑", "Edit", new { id=item.uid }) %>  |
<%= Html.ActionLink("删除", "delete", new { id=item.uid })%>
            </td>
        </tr>
        <% } %>
    </table>
</fieldset>
```

现在运行系统，先登录系统再单击“用户管理”链接来查看用户列表，如图 14-11 所示。



图 14-11 查看用户列表

把鼠标移到列表的“编辑”链接上，会在地址栏中看到请求的类似 users/Edit/1 这样的地址。这里的数字表示当前用户的编号，Edit 说明要执行编辑动作。

在 UserController 中新建一个名为 Edit 的 Action，用于处理“编辑”被单击之后的业务逻辑，代码如下：

```
[Authorize]
public ActionResult Edit(int id)
{
    var u = db.users.First(d => d.uid == id);
    ViewData["jib"] = new SelectList(get jib(), "value", "key", u.jib);
    return View(u);
}
```

在这里 get_jib()是一个自定义方法，它的作用是返回一个能够在 HTML 下拉列表中显示的数据源。这个数据源其实是一个“键/值”对的字典，保存了用户的等级信息，代码如下：

```
private Dictionary<string, int> get jib()
{
    Dictionary<string, int> n = new Dictionary<string, int>();
    n.Add("管理员", 0);
    n.Add("vip 会员", 1);
    n.Add("普通会员", 2);
    return n;
}
```

接下来看看修改用户资料页面的布局代码，将下面的代码添加到 Users/Edit.aspx 文件中。

```
    <div>
        <%: Html.ActionLink("返回查看所有列表", "Index") %>
    </div>
    <%= Html.ValidationSummary("Edit was unsuccessful. Please correct the errors
and try again.") %>
    <% using (Html.BeginForm())
       {%>
    <%: Html.ValidationSummary(true) %>
    <fieldset id="admin_list">
        <legend>编辑</legend>
        <p>
            <%= Html.Hidden("uid") %>
        </p>
        <p>
            <label for="name">
                姓名:</label>
            <%= Html.TextBox("name") %>
            <%= Html.ValidationMessage("name", "*") %>
        </p>
        <p>
            <label for="pwd">
                密码:</label>
            <%= Html.Password("pwd") %>
            <%= Html.ValidationMessage("pwd", "*") %>
        </p>
        <p>
            <label for="jib">
                级别:</label>
            <%= Html.DropDownList("jib") %>
            <%= Html.ValidationMessage("jib", "*") %>
        </p>
        <p>
            <label for="email">
                E-mail:</label>
            <%= Html.TextBox("email") %>
            <%= Html.ValidationMessage("email", "*") %>
        </p>
        <p>
            <label for="qq">
                QQ:</label>
            <%= Html.TextBox("qq") %>
            <%= Html.ValidationMessage("qq", "*") %>
        </p>
        <p>
            <input type="submit" value="修改" />
        </p>
    </fieldset>
    <% } %>
```

从上述代码可以看到，对于这样的表单在本节前面就曾使用过。因此这里不作过多的介绍，但是要注意数据获取的方式。

最后，创建一个同名的 Action 来接收 POST 提交的数据，使修改后的资料生效。具体实现

代码如下：

```
[Authorize]
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(int id, users u)
{
    try
    {
        var u1 = db.users.First(d => d.uid == id);
        string pwd = u1.pwd;
        UpdateModel(u1, new[] { "name", "pwd", "email", "qq" });
        if (u.pwd.Length > 0)
            u1.pwd = SqlHelper.md5(u.pwd);
        else
            u1.pwd = pwd;
        db.SubmitChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

再次运行系统，通过用户列表中的“编辑”链接进入编辑用户资料页面，效果如图 14-12 所示。编辑完成后，单击【修改】按钮保存并跳转到 index 页面。

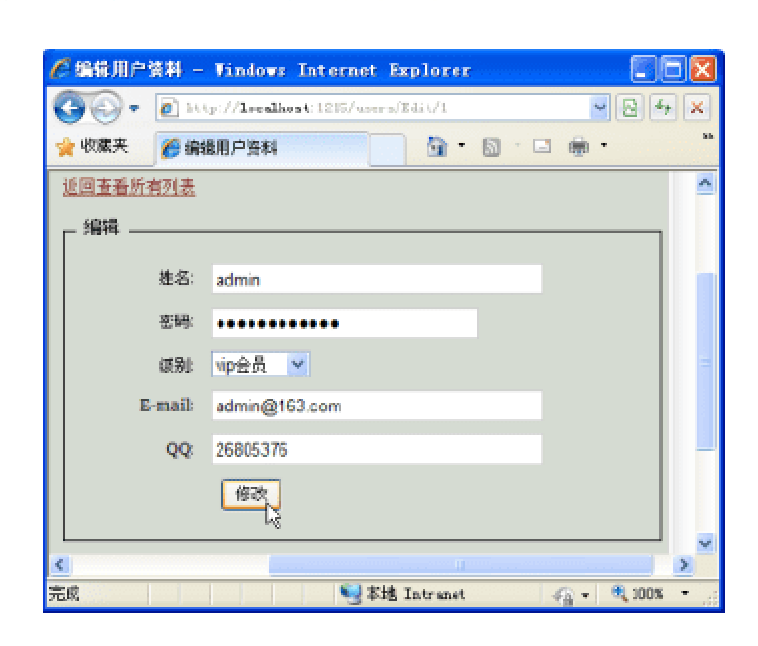


图 14-12　编辑用户资料

在用户列表中还可以看到“删除”链接，单击它之后将执行 Delete 动作，并传递一个整型参数。该参数用来标识要删除用户的编号，具体实现代码如下：

```
[Authorize]
[AcceptVerbs(HttpVerbs.Delete)]
public ActionResult Delete(int id)
{
    var u = db.users.First(d => d.uid == id);
    db.users.DeleteOnSubmit(u);
    db.SubmitChanges();
    return RedirectToAction("index");
}
```



编辑操作的 Edit 动作使用的是 HttpVerbs.Post，而在删除的 Delete 动作中使用的是 HttpVerbs.Delete。

14.6 后台管理模块

后台管理主要体现在栏目和文章两个方面，另外还有一个跟全局有关的配置信息。后台管理模块采用了 admin.Master 作为母版页，下面介绍具体的实现过程。

14.6.1 栏目管理

所谓栏目管理，就是为文章定义一些类别，然后在添加文章的时候进行选择，这些数据保存在 cls 表中。

在项目中新建 clsController，并添加默认的 Index 动作和视图文件，然后在 Index.aspx 中制作栏目管理的布局，代码如下：

```
    <h2>栏目管理</h2>
    <fieldset>
        <legend>栏目管理</legend>
        <p><a href="#" onclick="add(0)">添加一级栏目</a>
        </p>
        <table style="width: 95%" id="tb">
        </table>
        <div id="add" style="display: none">
            栏目名称: <input type="text" id="adStr" name="adStr"
style="width: 100px;" /> 
            <input type="button" onclick="add_sub()" value="添加" />
        </div>
        <div id="edit" style="display: none">
            栏目名称: <input type="text" name="itStr" id="itStr"
style="width: 100px;" /> 
            <input type="button" onclick="edit_sub()" value="修改" />
        </div>
</fieldset>
```

由于这里采用了 Ajax 无刷新技术，页面无须执行跳转，因此只用一个文件即可实现管理功能。下面添加一些 jQuery 代码，使页面加载完成后显示栏目列表，实现代码如下：

```
    <script type="text/javascript">
        $(document).ready(function () {
            list();
        });
        function list() {
            var str = "<tr><th style=\"width:50px\">编号</th><th>栏目名称</th><th
style=\"width:100px\">操作</th></tr>";
            $.getJSON("/cls/list",
            function (data) {
                $.each(data, function (i, d) {
                    str += "<tr><td>" + d.cid + "</td><td>" + d.name + "</td><td><a
href=# onclick=\"add(" + d.cid + ")\">添加</a> <a href=# onclick=\"edit("
+ d.cid + ")\">修改</a>  <a href=# onclick=\"dele(" + d.cid + ")\">
删除</a> </td></tr>";
```

```
            });
            $("#tb").html("");
            $("#tb").append(str);
        });
    }
</script>
```

如上述代码所示，list()函数负责以 JSON 形式将结果以表格形式显示出来，它请求的是/cls/list 动作。

为 clsController 添加 list 动作并编写如下代码：

```
        [Authorize]
        public JsonResult list()
        {
            Response.Cache.SetCacheability(HttpCacheability.NoCache);
            var d1 = db.cls.OrderBy(d => d.demo);
            List<cls> d2 = new List<cls>();
            foreach (var d in d1)
            {
                cls d3 = new cls();
                d3.cid = d.cid;
                string k = "";
                for (var i = 0; i < d.demo.Length / 2 - 1; i++)
                {
                    k += "  ";
                }
                d3.name = k + d.name;
                d2.Add(d3);
            }
            JsonResult js = Json(d2, JsonRequestBehavior.AllowGet);
            return js;
        }
```

我们知道，在栏目管理中需要实现添加、修改和删除栏目操作。这一步，我们来添加执行这些操作所需的客户端 jQuery 代码，主要代码如下：

```
        var id = 0;
        function add(id1) {
            id = id1;
            dialog("添加栏目", "id:add", "250px", "auto", "text");
        }
        function edit(id1) {
            id = id1;
            dialog("修改栏目", "id:edit", "250px", "auto", "text");
        }
        function add sub() {
            var addStr = $("[name='adStr']").eq(1).val();
            $.post("/cls/Create", { cid: id, name: addStr },
           function (data) {
              list();
           });
        }
        function edit sub() {
            var editStr = $("[name='itStr']").eq(1).val();
            $.post("/cls/edit", { cid: id, name: editStr },
```

```
            function (data) {
                list();
            });
        }
        function dele(d1) {
            $.post("/cls/delete", { cid: d1 },
            function (data) {
                list();
            });
        }
```

图 14-13 为添加栏目时的效果，这里的浮动对话框调用了第三方的 JavaScript 插件。

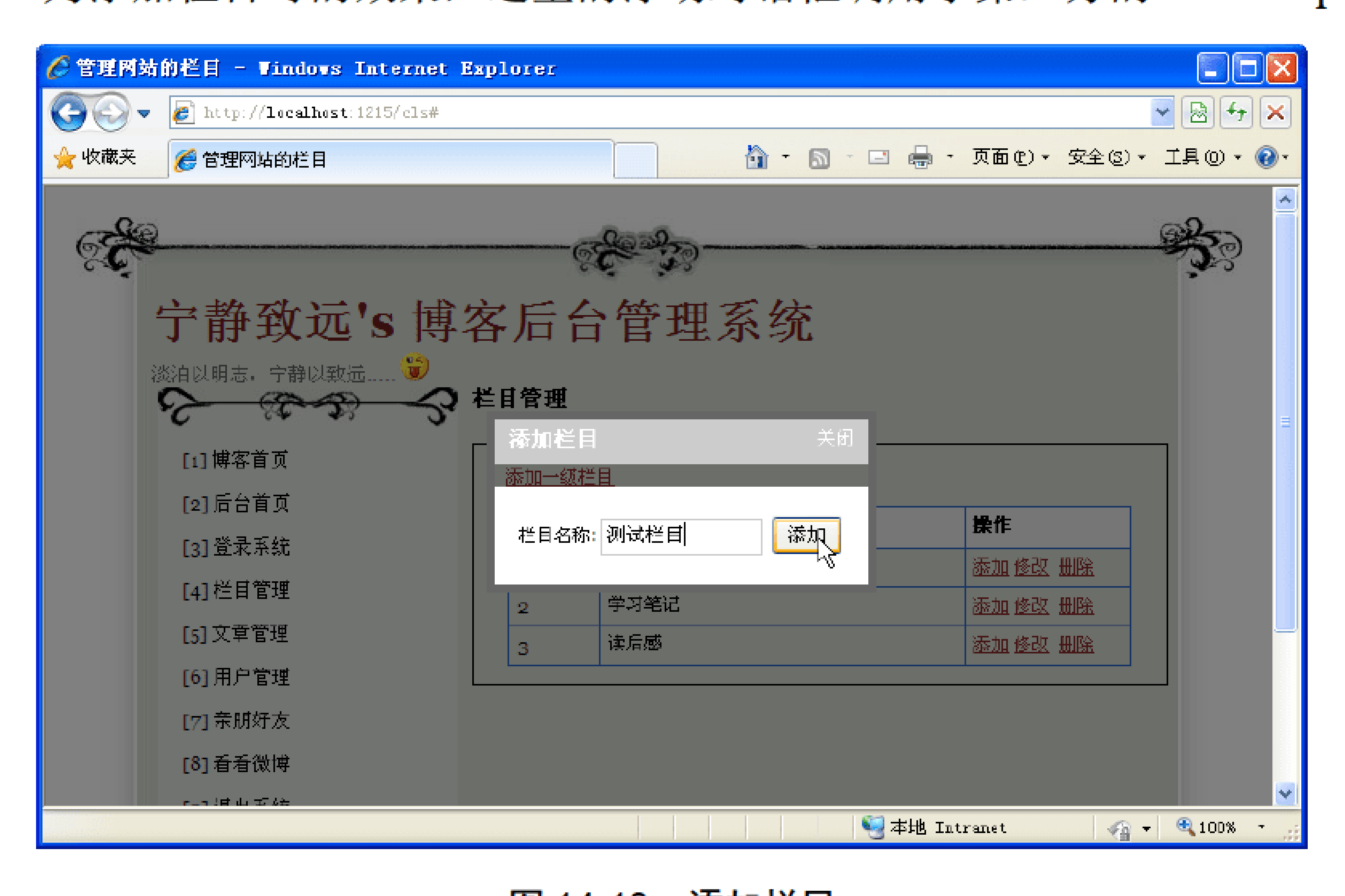


图 14-13　添加栏目

在文本框中输入一个新名称，再单击【添加】按钮。【添加】按钮执行的是 add_sub()函数，该函数将新栏目名称发送到 cls/create 动作。该动作的实现代码如下：

```
        [Authorize]
        public ContentResult Create(int cid, string name)
        {
            cls c1 = new cls();
            c1.cid = SqlHelper.get_ID("cls", "cid");
            c1.name = Server.UrlDecode(name);
            var f = from d in db.cls
                    where d.cid == cid
                    select d;
            if (f.Count() > 0)
            {
                cls f1 = f.First();
                c1.demo = f1.demo + c1.cid + "|";
            }
            else
            {
                try
                {
                    var f2 = (from d in db.cls
```

```
                                where d.demo.Length == 2
                                orderby d.demo descending
                                select d).First();
                    c1.demo = (Convert.ToInt32(f2.demo.Substring(0, 1)) +
1).ToString() + "|";
                }
                catch
                {
                    c1.demo = "1|";
                }
            }
            db.cls.InsertOnSubmit(c1);
            db.SubmitChanges();
            return Content(name);
        }
```

Create 动作将数据写入 cls 表之后，重新调用 list()函数来读取并加载最新的栏目列表。此时，便可以看到新增的栏目信息，如图 14-14 所示。

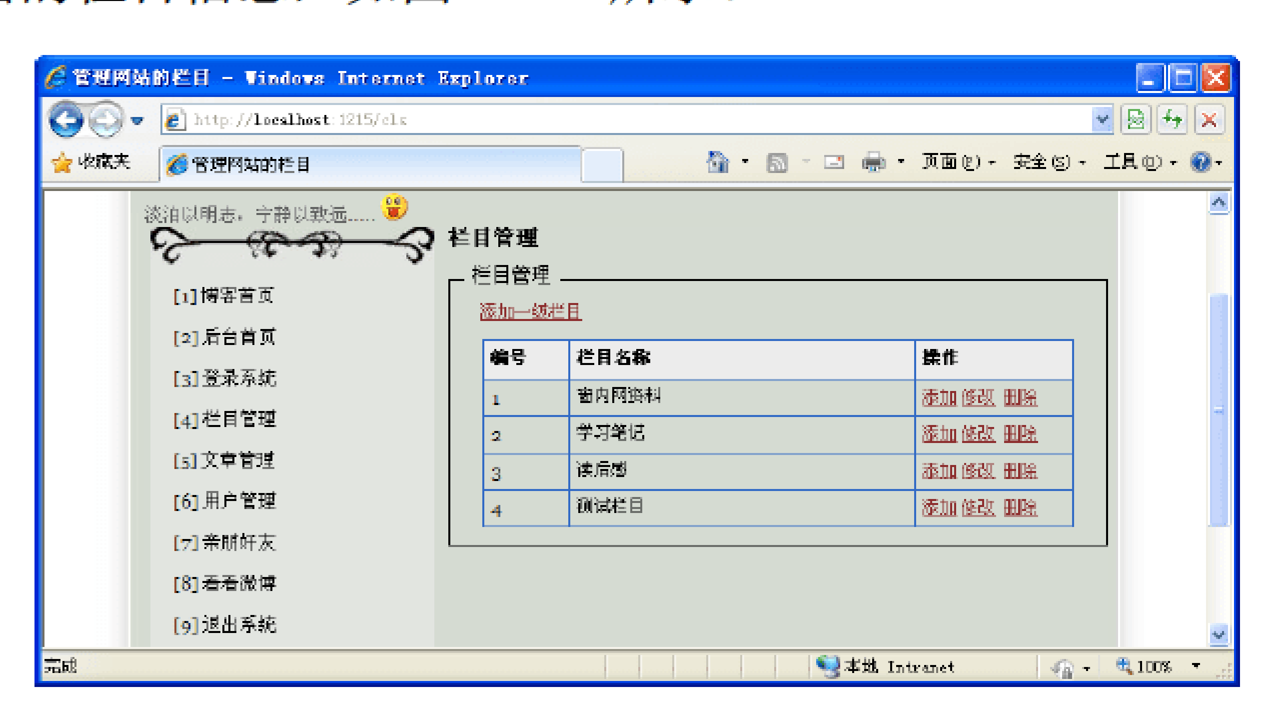


图 14-14　查看所有栏目

14.6.2　文章管理

文章管理使用的是 NewsController，在默认的 Index 动作中会显示所有的文章，并提供“修改”和“删除”两个操作链接，页面效果如图 14-15 所示。



图 14-15　查看所有文章

下面介绍文章管理功能的具体实现过程。

1. 文章列表

在 NewsController 中添加 Index 动作，然后添加如下的代码，实现从 News 表中获取相关数据并传递到 Index.aspx 视图页面。

```
[Authorize]
public ActionResult Index(string drop, string page)
{
    var d2 = droplist();
    ViewData["drop"] = new SelectList(d2, "cid", "name");
    var n = from d in db.news
            select d;
    if (drop != null)
    {
        n = from d in n
            where d.cid == Convert.ToInt32(drop)
            select d;
    }
    n = n.OrderByDescending(d => d.ndate);
    int size = 10;
    if (page != null)
    {
        ViewData["page"] = SqlHelper.Pager(Request.RawUrl.Substring(0,
Request.RawUrl.IndexOf("?")), Convert.ToInt32(page), n.Count(), size, 3);
        n = n.Skip((Convert.ToInt32(Request["page"]) - 1) *
size).Take(size);
    }
    else
    {
        ViewData["page"] = SqlHelper.Pager(Request.RawUrl, 1, n.Count(),
size, 3);
        n = n.Skip((1 - 1) * size).Take(size);
    }
    return View(n);
}
```

上述代码使用了 Linq 的分页机制，每页显示 10 条记录。其中的 droplist()是一个泛型集合，它的数据可以由我们自己定义，用于以下拉列表框的形式显示所有栏目名称。

以下是 droplist()方法的实现代码：

```
private List<cls> droplist()
{
    var d1 = db.cls.OrderBy(d => d.demo);
    List<cls> d2 = new List<cls>();
    foreach (var d in d1)
    {
        cls d3 = new cls();
        d3.cid = d.cid;
        string k = "";
        for (var i = 0; i < d.demo.Length / 2 - 1; i++)
        {
            k += "  ";
        }
        d3.name = k + d.name;
```

```
                d2.Add(d3);
            }
            return d2;
        }
```

有了数据源，接下来我们看看图 14-15 所示效果的布局是如何制作的。在 news/index.aspx 中继承 Admin.Master 母版页，然后用 foreach 遍历数据源，最终以表格形式显示出来。部分代码如下：

```
    <p>
            <%= Html.ActionLink("添加一篇新的文章", "Create") %>
        </p>
    <fieldset>
        <legend>文章列表</legend>
        <%=ViewData["d"] %>
        <table style="width:100%">
            <tr>
                <td colspan="6">
                    <%Html.BeginForm(); %>
                    <%=Html.DropDownList("drop") %>
                    <%= Html.SubmitButton("搜索") %>
                    <%Html.EndForm(); %>
                </td>
            </tr>
            <tr>
                <th>编号</th>
                <th>标题</th>
                <th>用户</th>
                <th>栏目</th>
                <th>发布时间</th>
                <th> </th>
            </tr>
            <% foreach (var item in Model)
               { %>
            <tr>
                <td><%= Html.Encode(item.nid) %></td>
                <td><%= Html.Encode(item.title) %>/td>
                <td> <%= Html.Encode(item.users.name) %> </td>
                <td>  <%= Html.Encode(item.cls.name) %> </td>
                <td> <%= Html.Encode(String.Format("{0:g}", item.ndate)) %>
</td>
                <td>
                    <%= Html.ActionLink("修改", "Edit", new { id=item.nid }) %>

                    <%= Html.ActionLink("删除", "delete", new { id=item.nid })%>
                </td>
            </tr>
            <% } %>
            <tr>
                <td colspan="6" class="page">
                    <%=ViewData["page"] %>
                </td>
            </tr>
        </table>
    </fieldset>
```

2. 发表文章

在博客系统中发表文章是博客会员经常要完成的动作之一。该操作映射到数据库，就是将博客会员所输入的信息插入数据库表中。要完成该操作，浏览者需要进行注册，注册完成后进行登录就可以进行文章添加操作了。

发表文章时可以指定新文章的标题、所属栏目、内容、摘要以及标签信息。图 14-16 为发表文章的页面效果。

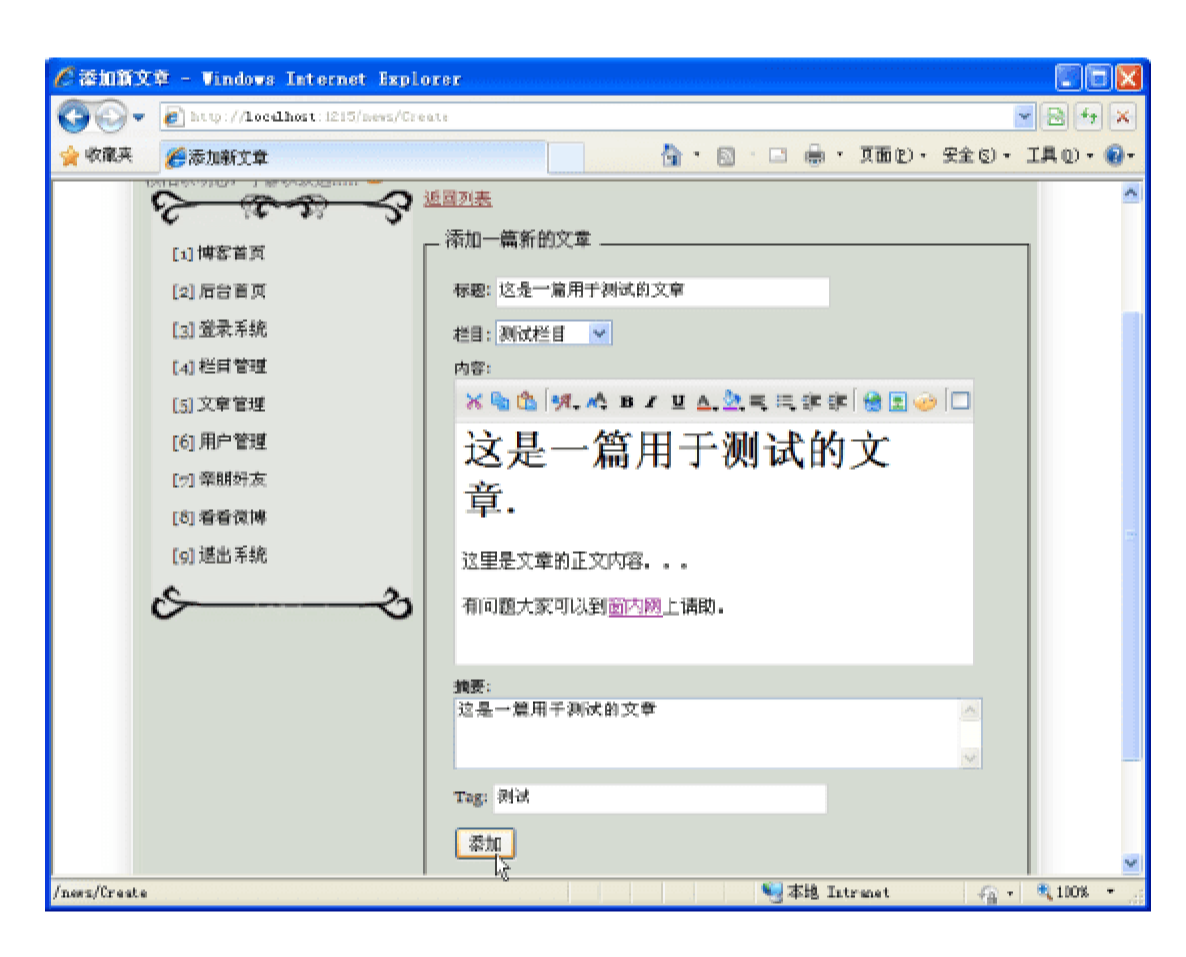


图 14-16　发表文章

可以看到，请求的 Action 是 news/Create。为这一步创建一个 Action 并添加如下代码：

```
        [Authorize]
        public ActionResult Create()
        {
            var d2 = droplist();
            ViewData["cid"] = new SelectList(d2, "cid", "name");
            news d = new news();
            return View(d);
        }
```

上述代码中的 ViewData["cid"]将作为【栏目】下拉列表框的数据源。创建对应的视图 news/create.aspx 文件，并根据图 14-16 所示的效果制作布局，最终实例代码如下：

```
    <div>
        <%=Html.ActionLink("返回列表", "Index") %>
    </div>
     <%= Html.ValidationSummary("Create was unsuccessful. Please correct the
errors and try again.") %>
    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>
         <fieldset>
            <legend>添加一篇新的文章</legend>
            <p>
                <%= Html.Hidden("nid") %>
            </p>
            <p>
                <label for="title">标题:</label>
```

```
                <%= Html.TextBox("title") %>
                <%= Html.ValidationMessage("title", "*") %>
            </p>
            <p>
                <label for="cid">栏目:</label>
                <%= Html.DropDownList("cid") %>
            </p>
            <p>
                <label for="cont">内容:</label>
                <%= Html.TextBox("cont",null, new {@style="display:none" })%>
                <iframe src="/content/editor/editor.html?id=cont"
frameborder="0" scrolling="no"  style="width:400px" height="220"></iframe>
                <%= Html.ValidationMessage("cont", "*") %>
            </p>
            <p>
                <label for="summary">摘要:</label>
                <%= Html.TextArea("summary", new
{@style="width:400px;height:50px;" })%>
                <%= Html.ValidationMessage("summary", "*") %>
            </p>
            <p>
                <label for="tag">Tag:</label>
                <%= Html.TextBox("tag") %>
                <%= Html.ValidationMessage("tag", "*") %>
            </p>
            <p>
                <input type="submit" value="添加" />
            </p>
        </fieldset>
    <% } %>
```

在这里要注意，由于在输入文章内容时使用了第三方的文本编辑器，为了保证它能够正常工作，必须将它放到能够访问的目录下，本实例中是 content 目录。

最后，我们来看看在上述表单中单击【添加】按钮之后执行的代码。此时，仍然是请求 Create 动作，所不同的是以 POST 方式执行，它获取用户的输入并向 News 表中插入一条新记录。实现代码如下：

```
        [Authorize]
        [ValidateInput(false)]
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Create(news n)
        {
            n.nid = SqlHelper.get ID("news", "nid");
            n.uid = 1;
            n.pic = "";
            n.cont = HttpContext.Server.HtmlEncode(n.cont);
            n.ndate = DateTime.Now;
            db.news.InsertOnSubmit(n);
            db.SubmitChanges();
            string[] tags = n.tag.Split(',');
            foreach (var t in tags)
            {
                var t2 = db.tag.Where(d => d.name == t);
                if (t2.Count() == 0)
```

```
            {
                tag tag1 = new tag();
                tag1.name = t;
                tag1.tid = SqlHelper.get ID("tag", "tid");
                tag1.sum = 1;
                db.tag.InsertOnSubmit(tag1);
            }
            else
            {
                var t3 = t2.First();
                t3.sum += 1;
            }
            db.SubmitChanges();
        }
        var d2 = droplist();
        ViewData["cid"] = new SelectList(d2, "cid", "name");
        return View();
    }
```



修改文章和删除文章的实现方法与栏目管理类似，这里就不再详细介绍了。修改文章对应的是 Edit 动作，删除文章对应的是 Delete 动作。

14.6.3 全局信息配置

所谓全局信息配置，就是指将网站的各种基本信息保存在数据库中，然后在各个页面里调用它。如果某项信息(例如联系方式)需要变化，只需更新一下数据，整个网站中的所有引用页都将更新，非常方便。

在本实例中，用户登录之后将会跳转到此页面，如图 14-17 所示。

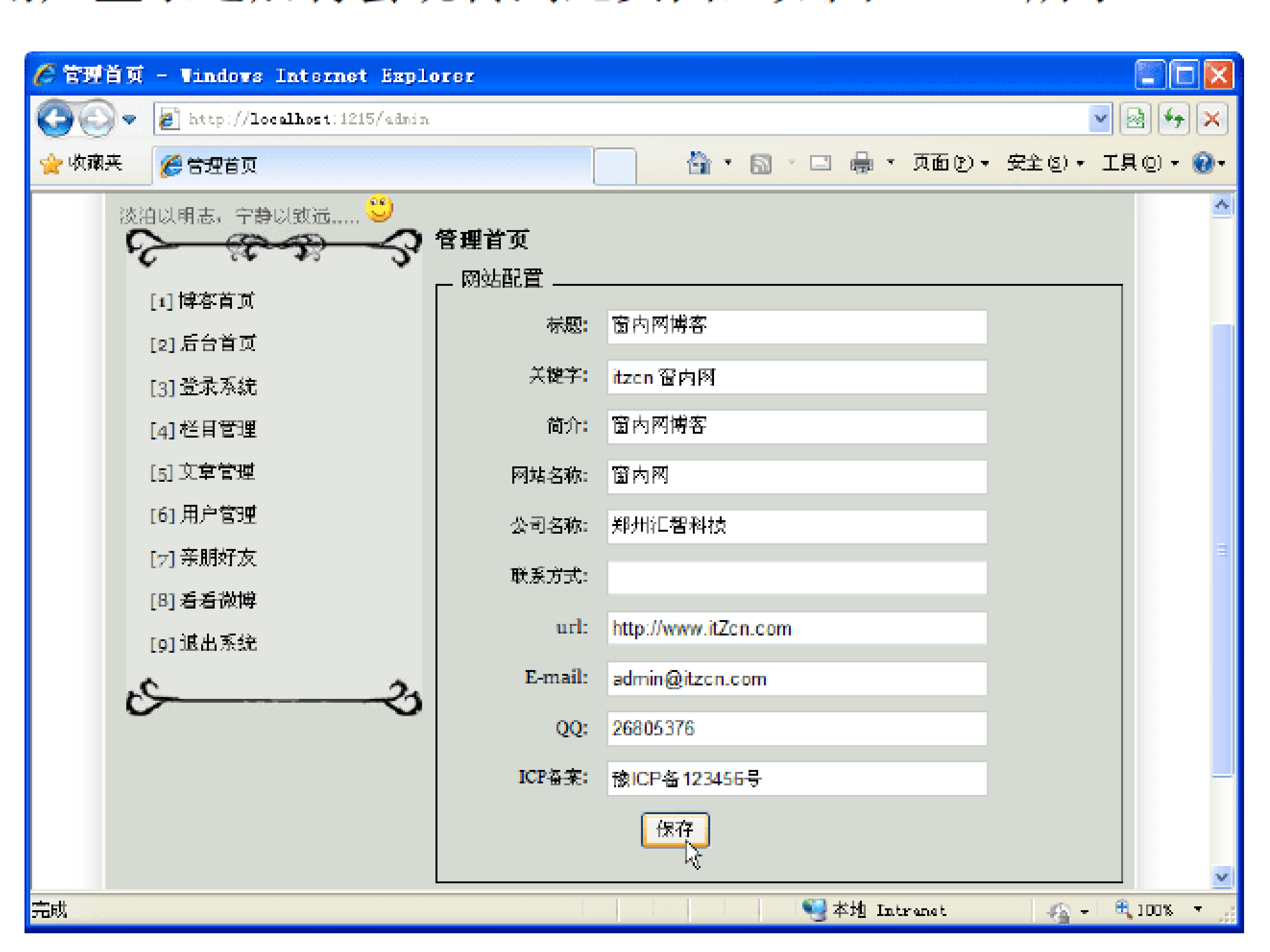


图 14-17　全局信息配置

可以看到，此页面使用的是 AdminController，默认的 Action 是 Index，代码如下：

```
    [Authorize]
    public ActionResult Index()
    {
```

```
            var d = db.she.First();
            return View(d);
        }
```

对应的视图文件是 Admin/Index.aspx，数据来自 she 表。这里的具体布局代码就不再给出，读者可以根据前面的知识自己做一下，也可以参考源代码。

最后来看看它是如何接收数据的。同样，与前面的过程类似，利用 Linq To Sql 中的 UpdateModel()方法即可更新模型到数据库。

```
        [Authorize]
        [AcceptVerbs(HttpVerbs.Post)]
        public ActionResult Index(int id, FormCollection s1)
        {
            var s2 = db.she.First(d => d.id == id);
            UpdateModel(s2, new[] { "title", "keywords", "description",
"sitename", "company", "tel", "url", "email", "qq", "beian" });
            db.SubmitChanges();
            var d2 = db.she.First();
            return View(d2);
        }
```

14.7　总　　结

博客以其可以充分利用网络互动、更新及时的特点，让用户最快获取最有价值的信息与资源，并可以发挥无限的表达力，及时记录和发布生活故事、新闻线索，以及与其他博友进行深度交流、沟通，发挥着越来越广泛的作用。

本章以我们非常熟悉的博客为话题，逐步展开对需求分析、功能分析、数据库设计、创建 MVC 项目以及实现各个功能等一系列过程的讲解。

采用 ASP.NET MVC 架构开发整个实例的流程非常清晰，避免了重复开发。例如，使用母版页有效降低 HTML 的工作量，实现快速更新。另外借助 Linq To Sql 实体，在整个 MVC 项目中不需要 Models 层就完全能够很好地工作。

最后要提示读者，在使用第三方文本编辑器时，可能会抛出“输入的 HTML 代码有潜在危险”的提示。解决的办法是在 Web.config 的 system.web 下添加如下代码：

```
<httpRuntime requestValidationMode="2.0" />
```

之后设置 pages 的 validateRequest 属性为 false 即可。



第 15 章　通讯录系统

内容摘要

近年来，随着电子产品彼此间的竞争日趋激烈，信息技术在企业的发展中占据着越来越重要的地位。在电子产品的系统设计上，电子通讯录的设计已经成为不可或缺的一部分，为各种信息的查询工作提供了重要的依据。

通讯录软件设计的灵感来源于生活和工作中的需求。如今，随着社会的飞速发展，信息时代改变着人们的各种生活方式。人们的联系信息和联系方式变得复杂而多样化，以前所使用的各种电话簿、通讯本等小册子，由于查找不方便、功能单一等缺陷已经无法胜任它的“时代使命”，而现在各种手机、商务通内设的电话簿，尽管携带方便却又挥之不去其“记录量少，界面小，浏览不方便”的缺点。工作中看到有些人巧妙地利用 Excel 或者 Word 制表格来建立通讯录，每逢用时再打开，可是查找极其不方便，维护起来也麻烦。

通讯录系统设计，它的内容对于电子产品来说是至关重要的。通讯录系统能够为电子产品的使用者提供充足的信息和快捷的查询手段。用 ASP.NET MVC 框架构建的通讯录系统，能够实现对通讯录信息的增加、删除和编辑等操作。本系统设计合理、操作方便、运行稳定、功能完备，具有较高的使用价值。

学习目标

- 了解并掌握控制器与视图间的数据传递
- 了解如何规定页面的访问形式
- 了解并掌握 Authorize 过滤器的使用
- 了解并掌握 Html.ActionLink 方法的使用

15.1　系 统 分 析

通讯录系统是典型的信息管理系统(Management Information System，MIS)，其开发主要包括后台数据库的建立和维护以及前端应用程序的开发两个方面。对于前者要求建立起数据一致性和完整性强、数据安全性好的数据库；而对于后者则要求应用程序具备功能完备、易使用等特点。

通讯录系统的功能是管理自己的通讯录，要求能对通讯录中的记录信息进行增加、删除和编辑操作，能够浏览联系人的基本信息，可以查看及上传照片等。

15.1.1　开发及运行环境

本系统所使用的系统开发平台为 Windows XP，Web 服务器为 IIS，数据库服务器为 Microsoft SQL Server 2008，开发工具采用了 Visual Studio 2010，技术架构为 ASP.NET MVC 2。

15.1.2　功能模块设计

通讯录系统一般分为用户登录模块、用户管理模块、照片管理模块、权限分析模块、留言本管理模块等 5 个模块，如图 15-1 所示。

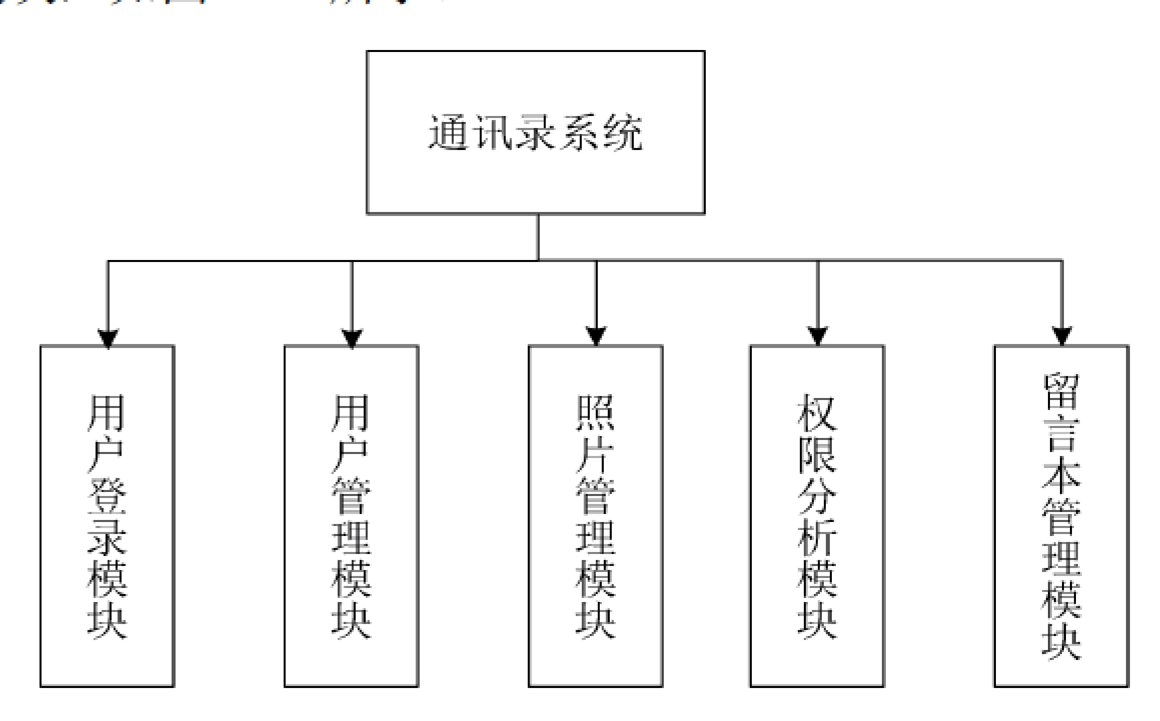


图 15-1　通讯录管理模块

- 用户登录模块：用户登录界面和修改密码。
- 用户管理模块：用户注册、删除和编辑。
- 照片管理模块：照片上传、删除和编辑。
- 权限分析模块：用户身份验证。
- 留言本管理模块：留言信息添加、删除和编辑。

15.1.3　数据库设计

数据库设计是根据用户的需求，在某一具体的数据库管理系统上，设计数据库的结构和建

立数据库的过程。本系统包含 3 张基本的数据表：User 表(用户表)、LeaveBook 表(留言本表)和 Img 表(照片表)。

1．User 表

User 表用于保存通讯录中的用户信息。例如用户名、密码等，具体如表 15-1 所示。

表 15-1　User(用户)表

字　段	数据类型	说　明
Id	int	用户编号
Name	nvarchar(30)	姓名
Password	varchar(35)	密码
Mobile	varchar(30)	手机号
Phone	varchar(30)	电话号码
QQ	varchar(30)	QQ 号码
Email	varchar(50)	电子邮箱
Company	nvarchar(40)	公司名称
Address	nvarchar(100)	联系地址
RegTime	datetime	注册时间
LastActiveTime	datetime	最后访问时间
LastActiveIp	varchar(30)	最后访问 IP
HeadImg	varchar(50)	头像
IsLock	bit	是否锁定

2．LeaveBook 表

LeaveBook 表用于保存用户的留言信息。例如标题、留言内容、留言用户等，具体如表 15-2 所示。

表 15-2　LeaveBook(留言本)表

字　段	数据类型	说　明
Id	int	留言编号
Title	nvarchar(100)	标题
[Content]	ntext	备注
AddTime	datetime	留言时间
Ip	varchar(30)	IP 地址
UserId	int	用户编号

3．Img 表

Img 表用于保存用户上传的照片信息。例如标题、上传者、上传时间等，具体如表 15-3 所示。

表 15-3　Img(照片)表

字　段	数据类型	说　明
Id	int	图片编号
Title	nchar(10)	标题
[Content]	ntext	备注
FileName	varchar(50)	上传者
AddTime	datetime	上传时间
UserId	int	用户编号

15.2　系统具体实现

通讯录系统主要包括以下几个功能模块。

- 用户登录模块　输入用户名和密码。
- 用户管理模块　添加、修改和删除用户。
- 通讯录信息添加模块　不重复添加。
- 通讯录信息删除模块　确认是否删除。
- 通讯录信息修改模块。
- 权限分析模块　用户身份验证。

15.2.1　用户登录模块

用户登录模块是用户进入后台管理界面的唯一通道，如果该用户没有注册，那么可以进入注册界面进行注册。用户登录模块的流程图如图 15-2 所示。

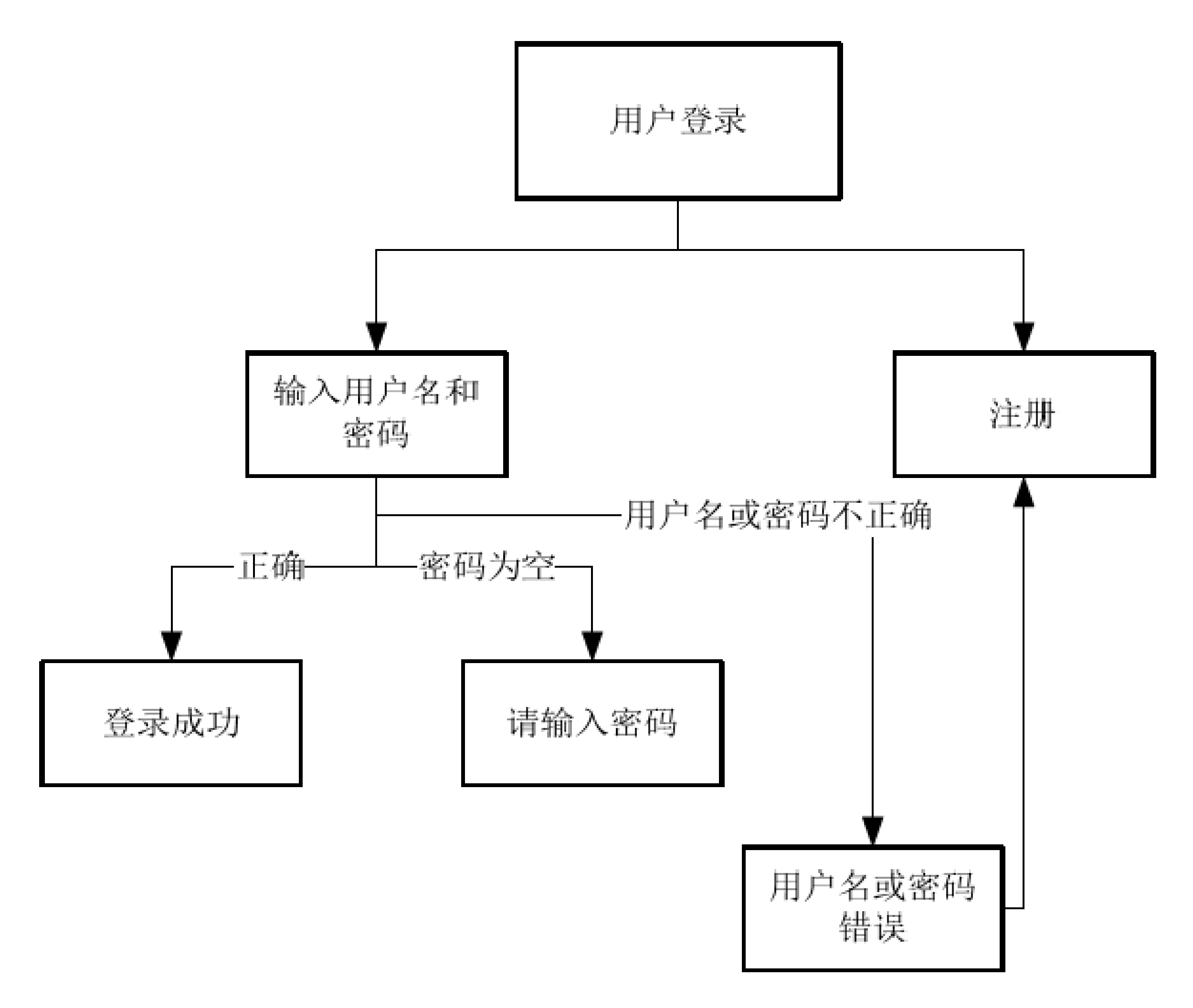


图 15-2　用户登录流程图

下面介绍用户登录模块的具体实现过程。

1．登录界面

登录界面用于用户登录本系统，有用户名和密码的可以直接登录。用户登录时，验证用户输入的用户名和密码是否正确，如果正确便登录成功；如果用户名或密码不正确，提示“用户名或密码错误”；如果密码为空，就提示“请输入密码”。

(1) 创建一个 ASP.NET MVC 2 Web 应用程序，名称为 MVC_addressBook。

(2) 在 Web.config 文件中设置数据库连接字符串，关键代码如下：

```
<add name="tongxunluSqlServer" connectionString="Data Source=.;Initial
Catalog=sanqi;User ID=sa;Password=123456"
providerName="System.Data.SqlClient"/>
```

(3) 在 Views/Shared 文件夹下，有一个母版页 Site.Master，下面设计这个母版页，作为整个系统的模板。body 元素里面的代码如下：

```
<div class="header">
    <h1>通讯录系统</h1>
</div>
<div id="logindisplay">
  <% Html.RenderPartial("LogOnUserControl"); %>        //引用用户控件
</div>
<div class="navigation">
   <%= Html.ActionLink("首页", "Index", "Home")%>
   <%= Html.ActionLink("照片", "Index", "Img")%>
   <%= Html.ActionLink("留言本", "Index", "LeaveBook")%>
   <%= Html.ActionLink("关于", "About", "Home")%>
    <div class="clearer"><span></span></div>
</div>
<div class="container">
   <asp:ContentPlaceHolder ID="MainContent" runat="server">

   </asp:ContentPlaceHolder>
</div>
```

引用 Content 文件夹下的 default.css 文件，效果如图 15-3 所示。



图 15-3　母版页设计图

(4) 在 Views/Account 文件夹下，应用程序自带了一个用户登录界面 LogOn.aspx，但是还

需要修改，页面代码如下：

```
 <h2>用户登录</h2>
   <p>
      请输入您的姓名和密码. 如果您没有账号请单击这里<%= Html.ActionLink("注册",
"Register") %> .
   </p>
   <%= Html.ValidationSummary(".") %>
<span style="color: Red;">
      <%=Html.Encode(TempData["message"]) %></span>
   <% using (Html.BeginForm()) { %>
      <div>
         <fieldset>
            <legend>用户登录</legend>
            <p>
               <label for="username">姓名:</label>
               <%= Html.TextBox("username") %>
               <%= Html.ValidationMessage("Name") %>
            </p>
            <p>
               <label for="password">密码:</label>
               <%= Html.Password("password") %>
               <%= Html.ValidationMessage("password") %>
            </p>
            <p>
               <%= Html.CheckBox("rememberMe") %> <label class="inline"
for="rememberMe">记住密码?</label>
            </p>
            <p>
               <input type="submit" value="登 录" />
            </p>
         </fieldset>
      </div>
   <% } %>
```

(5) 在 Models/Models 文件夹下添加一个用户类 User，用于记录用户信息，例如用户编号、用户名和密码等。

(6) 判断用户登录，在 Account 控制器下实现如下代码：

```
 [AcceptVerbs(HttpVerbs.Post)]
 public ActionResult LogOn(string userName, string password, bool rememberMe,
string returnUrl)
 {
    if (ValidateLogOn(userName, password))            //验证用户是否通过
    {
       if (AllService.IsLogin(userName, password))    //验证用户是否存在
       {
          FormsAuth.SignIn(userName, false);          //用户未登录
          if (!String.IsNullOrEmpty(returnUrl))
          {
             return Redirect(returnUrl);
          }
          else
          {
             return RedirectToAction("Index", "Home");//跳转到用户列表界面
```

```
            }
        }
        else
        {
            TempData["message"] = "用户名或密码错误";//提示信息
            return View();
        }
    }
    return View();
}
```

运行程序，效果如图 15-4 所示。



图 15-4　用户登录界面

2．注册界面

新用户注册，输入用户名和密码，单击【注册】按钮，判断用户是否存在。在 Account 控制器下注册的动作方法如下：

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Register(string Name, string Password)
{
    if (ValidateRegistration(Name, Password))               //验证用户是否通过
    {
        User user = new User { Name = Name, Password = Password, LastActiveIp
= "" };
        if (!AllService.IsExistUser(Name))                  //判断用户是否存在
        {
            if (AllService.RegUser(user))                   //注册用户
            {
                FormsAuth.SignIn(Name, false);              //用户未登录
                TempData["message"] = "恭喜，注册成功";      //提示信息
                return RedirectToAction("Index", "Home");   //页面跳转
            }
```

```
            else
            {
                ModelState.AddModelError(" FORM", "注册失败！");//提示注册失败
            }
        }
        else
        {
            ModelState.AddModelError(" FORM", "该姓名已经被注册！");
        }
    }
      return View();  //如果注册失败的话继续待在这个页面
  }
```

用户注册页面如图 15-5 所示。

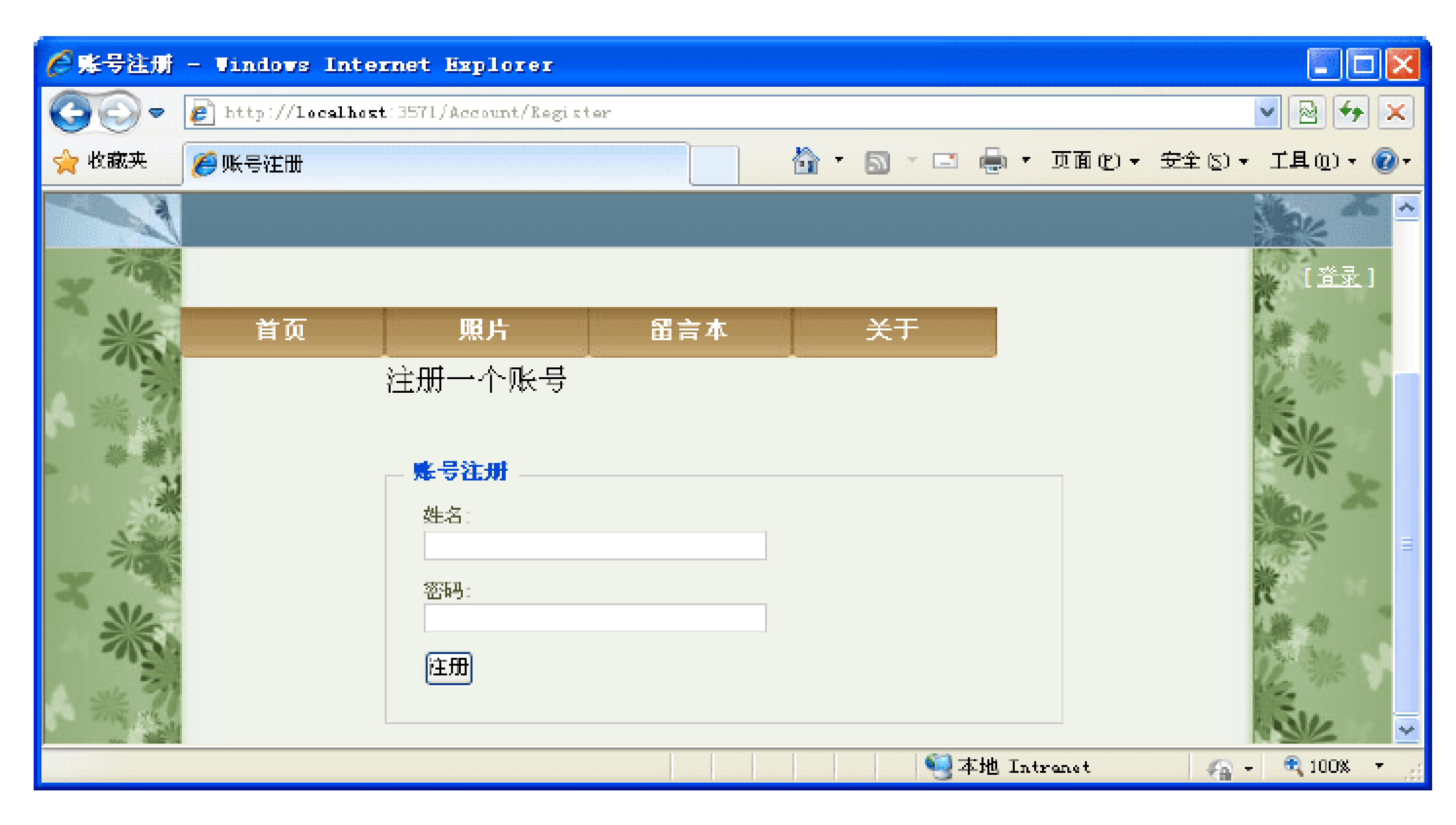


图 15-5　用户注册界面

3．修改密码

用户登录之后，可以修改自己的密码。在 Account 控制器下，修改密码的动作方法如下：

```
[Authorize]                                      //Authorize 过滤器
[AcceptVerbs(HttpVerbs.Post)]                    //设置以 POST 方式被访问
public ActionResult ChangePassword(string oldPassword, string newPassword,
string confirmPassword)
{
if (!ValidateChangePassword(oldPassword, newPassword, confirmPassword))
                                                 //验证密码不通过
{
     return View();
}
else
{
   try
    {
     if (AllService.ChangePassword(User.Identity.Name, oldPassword,
newPassword))
                                                 //验证密码通过
     {
       ModelState.AddModelError("_FORM", "密码修改成功.");
```

```
        return View();
      }
     else
      {
         ModelState.AddModelError(" FORM", "您输入的新密码出错.");
         return View();
      }
    }
     catch (Exception)
     {
        ModelState.AddModelError(" FORM", "您输入的密码出错或新密码非法.");
        return View();
     }
}
}
```

修改用户密码页面如图 15-6 所示。

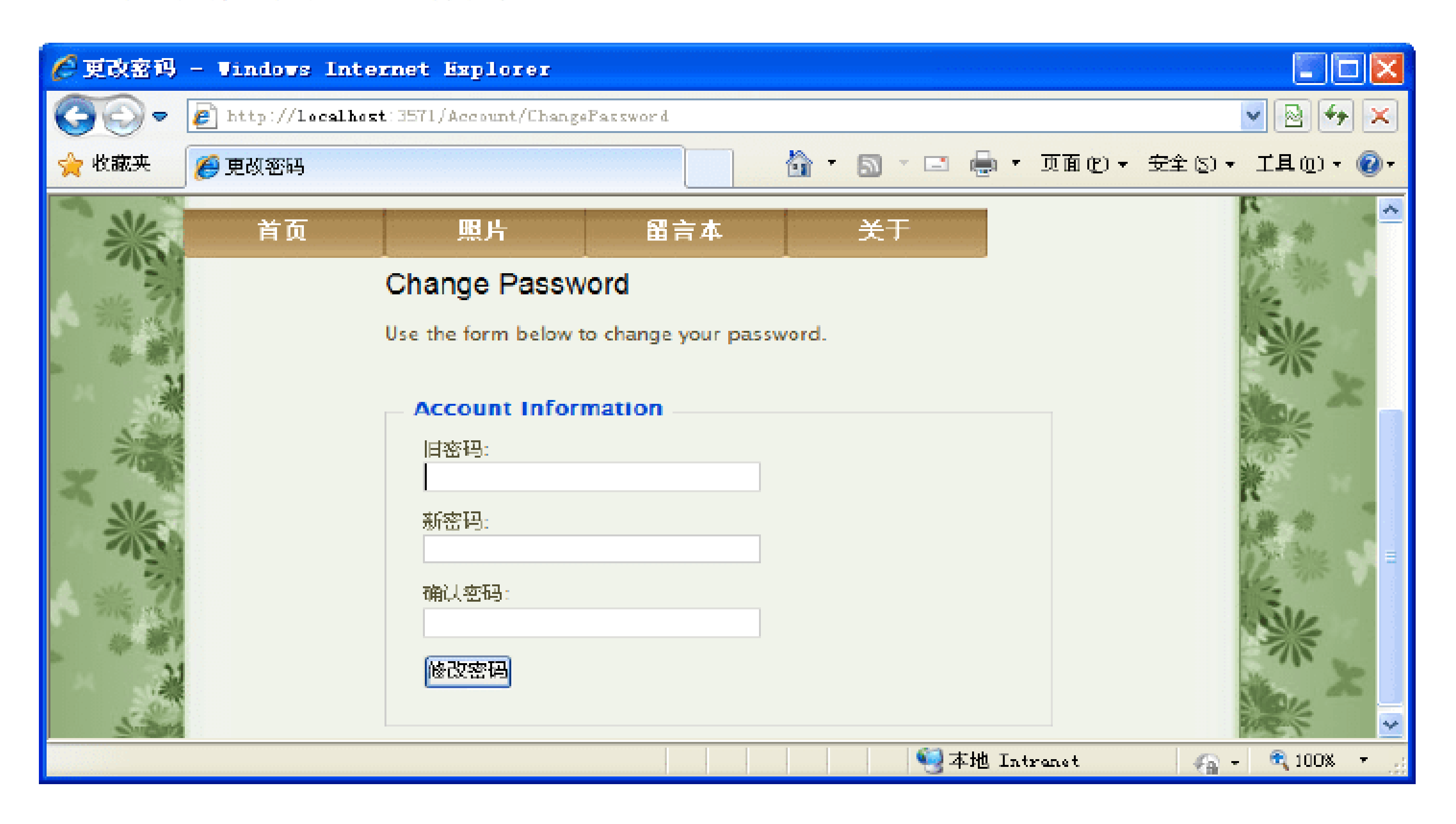


图 15-6　密码修改界面

15.2.2　用户管理模块

通讯录系统的用户管理模块包括编辑和删除用户信息，主要是对联系人的信息进行管理。用户登录成功之后，将进入用户管理界面。

下面介绍用户管理功能的具体实现过程。

(1) Home 控制器下的动作方法为 Index，用来从数据库中获取通讯录中联系人的信息，首先读取数据信息，并保存在集合列表中，实现代码如下：

```
public IList<User> GetUserList()
{
   IList<User> list = new List<User>();
   string sql = "select
[Id],[Name],[Mobile],[Phone],[QQ],[Email],[Company],[Address],[RegTime],[La
stActiveTime],[LastActiveIp],[HeadImg] from [User] order by [Id] desc";
   using (SqlDataReader rdr =
SqlHelper.ExecuteReader(SqlHelper.ConnectionString, CommandType.Text, sql,
null))
```

```
    {
        while (rdr.Read())
        {
        list.Add(new User { Id = rdr.GetInt32(0), Name = rdr.GetString(1), Mobile
        = rdr.GetString(2), Phone = rdr.GetString(3), QQ = rdr.GetString(4),
        Email = rdr.GetString(5), Company = rdr.GetString(6), Address = rdr.
        GetString(7), RegTime = rdr.GetDateTime(8), LastActiveTime = rdr.
        GetDateTime(9), LastActiveIp = rdr.GetString(10), HeadImg = rdr.
        GetString(11) });
        }
    }
    return list;
}
```

(2) 在 Home 控制器的 Index 动作方法中获取集合列表 list，并将它传递给视图，实现代码如下：

```
[Authorize]
public ActionResult Index()
{
    IList<User> userList = AllService.GetUserList();
    return View(userList);
}
```

(3) 在 Views/Home 文件夹下的 Index 视图中接收从 Home 控制器动作方法 Index 传过来的数据 list，数据模型类是 User，所以要将页面的 Inherits 属性值设置如下：

```
System.Web.Mvc.ViewPage<IEnumerable<MVC_addressBook.Models.Models.User>>
```

页面中接收用户数据的代码如下：

```
    <h2>用户列表</h2>
    <span style="color: Red;">
        <%=Html.Encode(TempData["message"]) %></span>
    <table>
        <tr>
            <th>
                Name
            </th>
            <th>
                Mobile
            </th>
            <th>
                Company
            </th><th>
            </th>
        </tr>
        <% foreach (var item in Model)
           { %>
        <tr>

            <td>
                <%= Html.Encode(item.Name) %>
            </td>
```

```
            <td>
                <%= Html.Encode(item.Mobile) %>
            </td>
            <td>
                <%= Html.Encode(item.Company) %>
            </td>
            <td>
                <%= Html.ActionLink("编辑", "Edit", new {  id=item.Id  }) %>
                |
                <%= Html.ActionLink("详细", "Details", new { id = item.Id })%>
            </td>
        </tr>
        <% } %>
    </table>
```

页面效果如图 15-7 所示。



图 15-7　用户管理列表

(4) 单击【编辑】链接，进入编辑用户信息界面，也就是 Views/Home/Edit.aspx 视图。同前面讲的一样，接收数据之前，首先将页面与对应的模型类 User 关联起来，设置代码如下：

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MVC_addressBook.Models.Models.User>" %>
```

显示要编辑的用户信息，例如显示用户编号，实现代码如下：

```
<div class="editor-label">
    <%: Html.LabelFor(model => model.Id) %>
</div>
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.Id) %>
    <%: Html.ValidationMessageFor(model => model.Id) %>
</div>
```

页面效果如图 15-8 所示。

单击【返回列表】按钮，就会回到用户列表的界面。使用 Html.ActionLink()方法可以实现页面跳转的效果，代码如下：

```
<%: Html.ActionLink("返回列表", "Index") %>
```

图 15-8　编辑用户信息

(5) 单击“详细”链接即可查看该用户的详细信息。例如查看用户编号和用户名等，可以用 Html.Encode()方法获取数据信息，代码如下：

```
<p>
    Id:
    <%= Html.Encode(Model.Id) %>
</p>
<p>
    Name:
    <%= Html.Encode(Model.Name) %>
</p>
```

页面效果如图 15-9 所示。

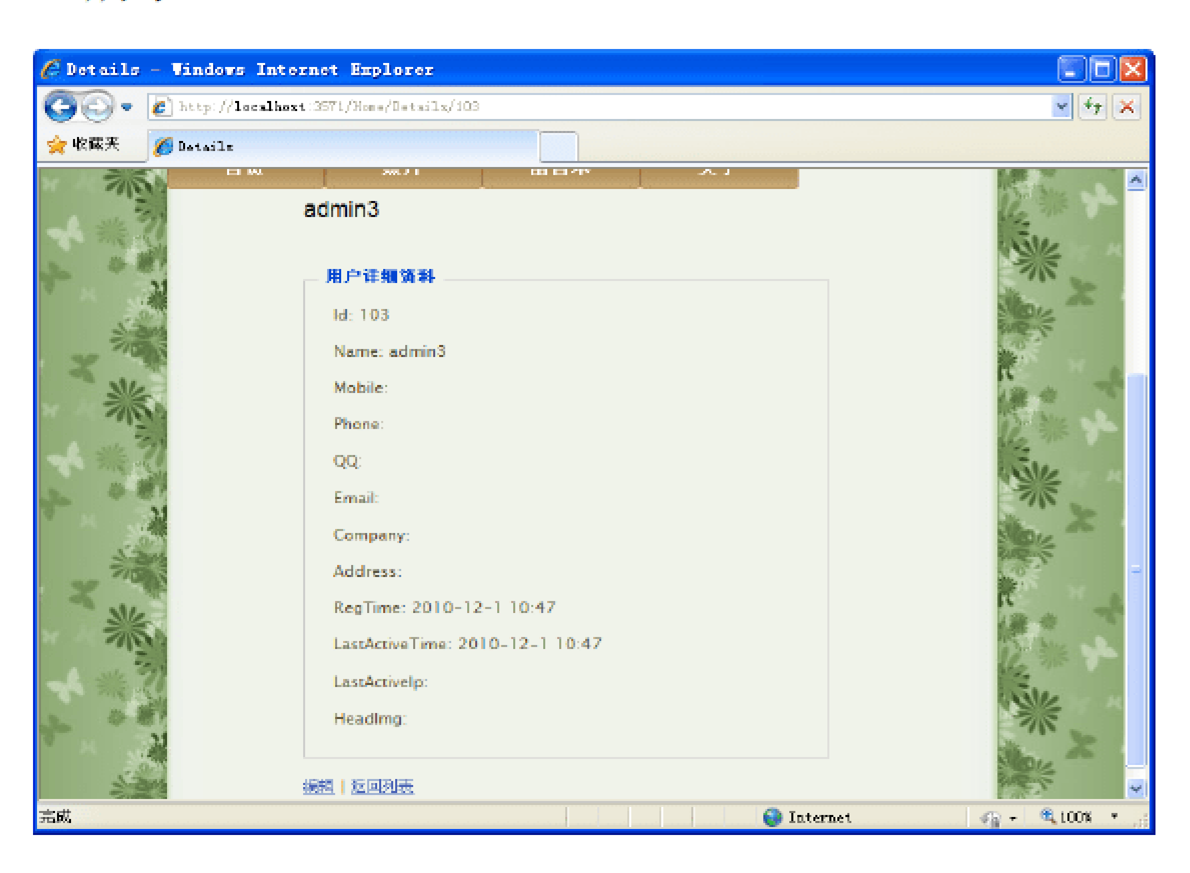


图 15-9　查看用户详细信息

15.2.3　照片管理模块

照片管理模块主要用于上传、编辑和删除用户照片。

下面介绍照片管理功能的具体实现过程。

(1) 在该系统中，Img 控制器下的 Index 动作方法用于获取照片的信息，并将这些信息显示在对应的 Index 视图中。动作方法 Index 的实现代码如下：

```
[Authorize]
public ActionResult Index(int? id)
{
    int rsCount = AllService.ImgCount();
    int PageCount = AllService.ImgPageCount();
    if (id == null || id < 1) id = 1;
    if (id > PageCount) id = PageCount;
    int PageNow = int.Parse(id.ToString());
    IList<Img> imgList = AllService.GetImgList(PageNow);
    ViewData["page"] = PageString.strPage(PageNow, PageCount, rsCount, "Index",
"Img");
    return View(imgList);
}
```

(2) 在 Index 视图中显示照片的所有数据，例如获取缩略图、标题等，页面实现代码如下：

```
<h2>照片列表</h2>
 <table>
     <tr>
         <th>
             缩列图
         </th>
         <th>
             标题
         </th>
         <th>
             上传时间
         </th>
         <th>
             上传用户
         </th>
         <th>
         </th>
     </tr>
     <% foreach (var item in Model)
        { %>
     <tr>
         <td>
             <img alt="" src='/Content/uploadImage/<%=item.FileName%>'
style="border:solid 2px #ccffff; width:100px; height:80px;"></img>
         </td>
         <td>
             <%= Html.Encode(item.Title) %>
         </td>
         <td>
             <%= Html.Encode(String.Format("{0:g}", item.AddTime)) %>
         </td>
         <td>
             <%= Html.Encode(item.UserName) %>
         </td>
         <td>
```

```
                <%= Html.ActionLink("详情", "Details", new { id=item.Id })%>
            </td>
        </tr>
        <% } %>
    </table>
```

页面显示效果如图 15-10 所示。

(3) 单击“详情”链接，即可查看该条数据的详细信息，例如在页面中获取照片的标题，其实现代码如下：

```
<div class="display-label">Title</div>
<div class="display-field"><%: Model.Title %></div>
```

页面效果如图 15-11 所示。



图 15-10　照片列表　　　　图 15-11　查看照片详细信息

(4) 返回照片列表，单击列表下面的“我要上传图片”链接，即可进入上传图片界面——Create 视图，代码如下：

```
    <p>
    <%= ViewData["page"] %>
    </p>
    <p>
        <%= Html.ActionLink("我要上传图片", "Create") %>
    </p>Html.ActionLink("Back to List", "Index") %>
    </p>
```

(5) 在 Img 控制器下的 Create 动作方法，用于判断上传图片是否成功，验证用户上传照片的条件，其实现代码如下：

```
[Authorize]
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(HttpPostedFileBase uploadpic)
{
    try
    {
        if (uploadpic != null)
        {
            HttpPostedFileBase file = Request.Files[0];
            string FileName = UpLoadImg.UpLoadPic(file);
            if (FileName.IndexOf("错误") > -1) //出现错误时，alert 错误
            {
                ViewData["Error"] = FileName;
```

```
                return View();
            }
            else
            {
                Img img = new Img();
                img.Title = Request.Form["Title"];
                img.Content = Request.Form["Content"];
                img.FileName = FileName;
                img.UserId = AllService.GetUserId(User.Identity.Name);
                if (AllService.SaveImg(img))
               {
                   TempData["message"] = "上传图片成功";
                   return RedirectToAction("Index", "Img");
               }
                else
                {
                    ViewData["Error"] = "数据库写入失败";
                    return View();
                }
            }
        }
        else
        {
            ViewData["Error"] = "请选择图片";
            return View();
        }
        //return RedirectToAction("Index");
    }
    catch (Exception ex)
    {
        TempData["message"] = "非法输入或系统错误:" + ex.Message;
        return RedirectToAction("Error", "Home");
        //return View();
    }
}
```

上传图片页面如图 15-12 所示。

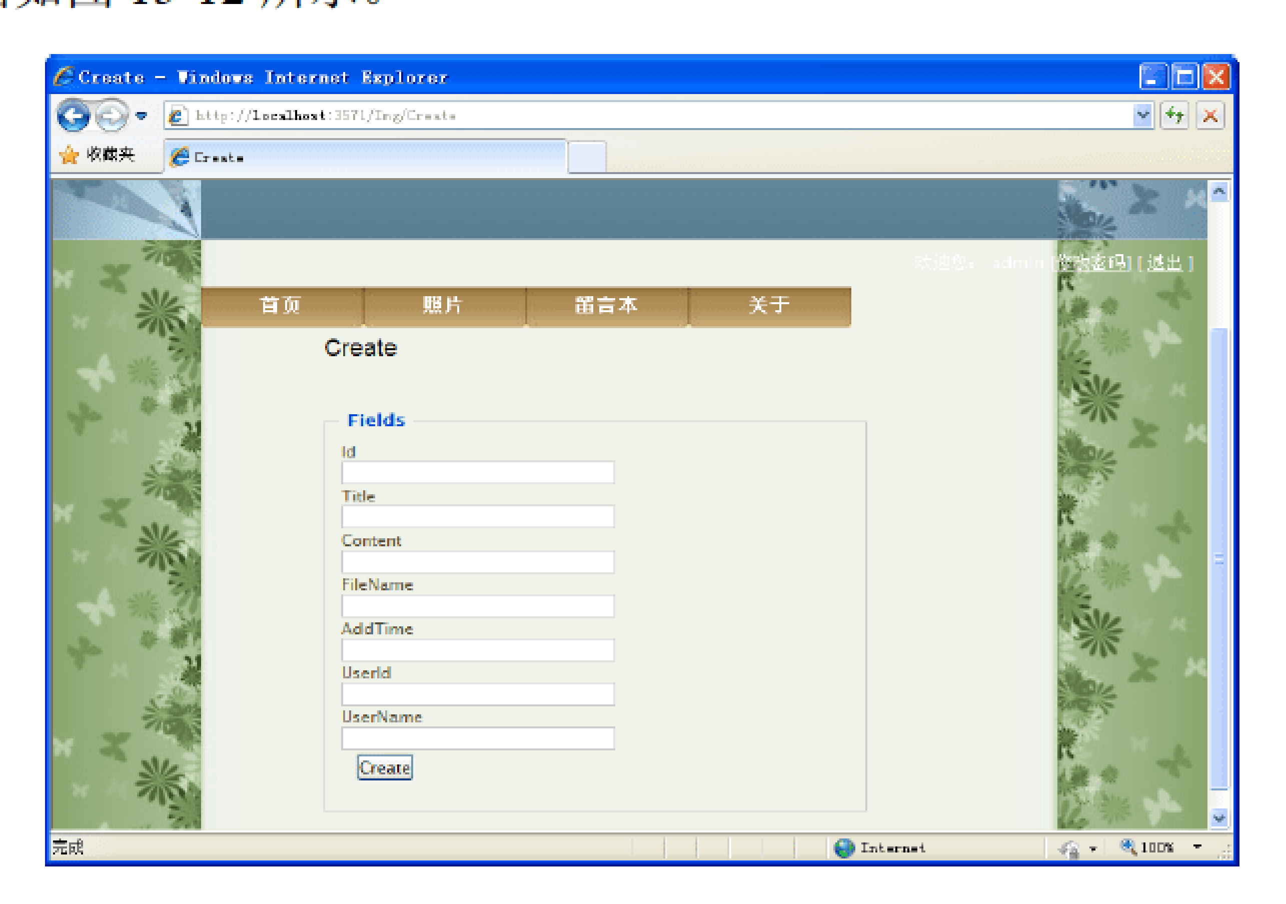


图 15-12　上传图片界面

15.2.4　权限分析模块

权限分析模块主要包括判断用户的访问权限，在 ASP.NET MVC 框架中，可以用过滤器来设置用户的访问权限。例如，在该系统中频繁使用的过滤器就是 Authorize 和 AcceptVerbs。

AuthorizeAttribute 是 ASP.NET MVC 默认的授权过滤器，可以使用它来限制对动作方法的访问。将该特性运用到控制器上，可以迅速将其运用到每个动作方法中。

在运用该过滤器时需要牢记如下内容。

- 在运用该特性时，可以指定一个逗号来划分角色(Role)或用户(User)的列表。如果指定了一个角色的列表，那么为了执行动作方法，用户必须是其中一个角色的成员。同样，如果指定了一个用户列表，那么当前用户的名称必须在该列表中。
- 如果没有指定任何角色或用户，那么为了调用动作方法，必须验证当前用户。这是阻止非验证用户访问特殊控制器动作的一个简单方法。
- 如果用户试图访问运用了该特性的动作方法且在授权检查中失败，那么过滤器将引发服务器返回一个 401 Unauthorized 的 HTTP 状态代码。
- 对于启用了表单验证且在 web.config 中指定了注册 URL 的情形，ASP.NET 将处理该响应代码，并将用户重新引向注册页面。这是 ASP.NET 已有的行为，对于 ASP.NET MVC 也不是新鲜事物。

在该系统中，Authorize 过滤器没有指定任何角色或用户，那么每个用户只可以对自己的信息进行操作，而对其他用户的信息只可以浏览，不可以修改。

AcceptVerbs 过滤器用来设置用户的访问方式，包括 POST 及 GET。

15.2.5　留言本管理模块

留言本管理模块主要包括对留言信息的添加、编辑和删除操作。在通讯录系统中，每个联系人都可以留言，这样所有的联系人都可以看到留言信息。

下面介绍留言本管理功能的具体实现过程。

(1) 从数据库中读取留言本里的所有留言信息，并保存在集合列表中，实现代码如下：

```
public IList<LeaveBook> GetLeaveBookList(int StartIndex, int EndIndex)
{
  IList<LeaveBook> list = new List<LeaveBook>();
  string sql = "select Id,Title,Content,AddTime,Ip,UserId,UserName from(select
row number() over(order by l.Id desc) as
row num,l.[Id],l.[Title],l.[Content],l.[AddTime],l.[Ip],l.[UserId],u.[Name]
as UserName from [LeaveBook] l join [User] u on u.[Id]=l.[UserId]) t where row num
between @StartIndex and @EndIndex";
  SqlParameter[] param = new SqlParameter[] { new SqlParameter("@StartIndex",
SqlDbType.Int), new SqlParameter("@EndIndex", SqlDbType.Int) };
  param[0].Value = StartIndex;
  param[1].Value = EndIndex;
  using (SqlDataReader rdr =
SqlHelper.ExecuteReader(SqlHelper.ConnectionString, CommandType.Text, sql,
param))
```

```
  {
    while (rdr.Read())
    {
     list.Add(new LeaveBook { Id = rdr.GetInt32(0), Title = rdr.GetString(1),
Content = rdr.GetString(2), AddTime = rdr.GetDateTime(3), Ip = rdr.GetString(4),
UserId = rdr.GetInt32(5), UserName = rdr.GetString(6) });
    }
  }
    return list;
}
```

(2) 在 Views/LeaveBook 文件夹下的 Index 视图中显示所有留言信息，实现代码如下：

```
<h2>留言本</h2>
    <span style="color: Red;">
    <%=Html.Encode(TempData["message"]) %></span>
    <table>
        <tr>
            <th>
                Id
            </th>
            <th>
                Title
            </th>
            <th>
                AddTime
            </th>
            <th>
                Ip
            </th>
            <th>
                UserName
            </th>
            <th>
            </th>
        </tr>
        <% foreach (var item in Model)
           { %>
        <tr>
            <td>
                <%= Html.Encode(item.Id) %>
            </td>
            <td>
                <%= Html.Encode(item.Title) %>
            </td>
            <td>
                <%= Html.Encode(String.Format("{0:g}", item.AddTime)) %>
            </td>
            <td>
                <%= Html.Encode(item.Ip) %>
            </td>
            <td>
                <%= Html.Encode(item.UserName) %>
            </td>
           <td>
                <%= Html.ActionLink("详情", "Details", new { id = item.Id })%>
```

```
            </td>
        </tr>
        <% } %>
    </table>
```

(3) LeaveBook 控制器下的 Index 动作方法用于获取数据库中所有的留言信息，并把这些信息传递到 Index 视图中，实现代码如下：

```
 [Authorize]
    public ActionResult Index(int? id)
    {
        int rsCount = AllService.LeaveBookCount();
        int PageCount = AllService.LeaveBookPageCount();
        if (id == null || id < 1) id = 1;
        if (id > PageCount) id = PageCount;
        int PageNow = int.Parse(id.ToString());
        ViewData["page"] = PageString.strPage(PageNow, PageCount, rsCount,
"Index", "LeaveBook");
        IList<LeaveBook> leaveBookList = AllService.GetLeaveBookList(PageNow);
        return View(leaveBookList);
   }
```

页面效果如图 15-13 所示。



图 15-13　留言本管理列表

(4) 单击“详情”链接，即可查看留言的详细信息。例如显示留言的标题、内容等，其实现代码如下：

```
<p>
    Title:
    <%= Html.Encode(Model.Title) %>
</p>
<p>
    Content:
    <%= Html.Encode(Model.Content) %>
</p>
```

页面效果如图 15-14 所示。



图 15-14　留言详情界面

(5) 单击“返回留言列表”，即可放弃留言页面返回留言列表页面，其实现代码如下：

```
<%=Html.ActionLink("返回留言列表", "Index") %>
```

(6) 在留言列表页面显示“我要留言”的链接，单击“我要留言”就会跳转到留言页面，里面有留言标题文本框、留言内容文本域和一个【提交】按钮。留言页面如图 15-15 所示。



图 15-15　留言页面

(7) 单击【提交】按钮，留言信息将被保存到数据库中，这个提交动作就是控制器动作 Create，实现代码如下：

```
[Authorize]
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(FormCollection colleciton)
{
    try
    {
        string userIp = GetClientIP();
        int UserId = AllService.GetUserId(User.Identity.Name);
```

```
        LeaveBook leaveBook = new LeaveBook();
        leaveBook.Ip = userIp;                              //ip 地址
        leaveBook.UserId = UserId;                          //用户编号
        leaveBook.Title = Request.Form["title"];            //获取留言标题
        leaveBook.Content = Request.Form["content"];        //获取留言内容
        if (ValidationLeaveBookCreate(leaveBook.Title, leaveBook.Content))
            //验证留言标题和内容是否符合条件
        {
            if (AllService.SaveLeaveBook(leaveBook))        //验证该标题是否存在
            {
                TempData["message"] = "留言成功!";
                return RedirectToAction("Index");
            }
            else
            {
                ModelState.AddModelError(" FORM", "留言失败.");
                return View();
            }
        }
        return View();
    }
    catch (Exception ex)
    {
        TempData["message"] = "非法输入:" + ex.Message;
        return RedirectToAction("Error", "Home");
        //return View();
    }
}
```

15.3 总　　结

日益繁多的人际交往使得我们很难记忆与每个人之间的联系方式，特别是对经常出差的人来说，更难，所以通讯录能够便捷地给我们带来所需要的相关信息。随着计算机的普及，人们的生活摆脱了传统式的记事本、电话簿，广泛地使用计算机来帮助人们记住这些事情，极其简便。那么通过使用通讯录系统，用户可以方便地查阅自己所需要的信息，而不用再颇费周折翻看那些繁琐的记事本。

通讯录系统是一个专门针对储存用户联系方式以及一些简单个人信息的系统，它提供了用户对众多客户、朋友和同事等个人信息的储存以及快速查阅的功能，大大节省了查找时间，减轻了用户记忆负担。

附录　习 题 答 案

第 1 章

一、填空题

(1) 控制器　(2) 业务逻辑　(3) MonoRail
(4) System.Web.Mvc　(5) Content　Scripts　(6) Global.asax

二、选择题

(1) C　(2) C　(3) A　(4) A　(5) C　(6) A　(7) A

第 2 章

一、填空题

(1) 参数　(2)Action　(3) System.Web.Routing　(4) 正则表达式

二、选择题

(1) D　(2) A　(3) A　(4) A

第 3 章

一、填空题

(1) Controller　(2) 公共　(3) ASP.NET MVC　专门
(4) 控制器　多　(5) Request.Form 集合　Request.QueryString 集合

二、选择题

(1) D　(2) B　(3) D　(4) B　(5) C

第 4 章

一、填空题

(1) 数据模型　(2) 业务模型　(3) 模型的重构　提高重用性

(4) 实体对象的数据保存　　(5) 模型

二、选择题

(1) B　(2) C　(3) D

第 5 章

一、填空题

(1) ascx　(2) Model　(3) View(zhangsan)　(4) TempData

二、选择题

(1) A　(2) A　(3) D　(4) D

第 6 章

一、填空题

(1) System.Web.Mvc　(2) BeginForm　(3) Encode()　(4) DropDownList()

二、选择题

(1) A　(2) B　(3) D　(4) A

第 7 章

一、填空题

(1) System.Web.UI.Page　(2) RepeatColumns　(3) DataBind()
(4) AlternatingItemTemplate

二、选择题

(1) D　(2) C　(3) A

第 8 章

一、填空题

(1) IviewEngine　(2) return View("Create", "UserMaster");　(3) $User.Name$

二、选择题

(1) B　(2) A　(3) D　(4) A

第 9 章

一、填空题

(1) 授权　(2) 异常　(3) OutputCache　(4) Controller　(5) ActionFilterAttribute

二、选择题

(1) B　(2) B　(3) C　(4) B　(5) C

第 10 章

一、填空题

(1) IexceptionFilter　(2) 虚　System.Web.Mvc.ExceptionContext
(3) CustomErrors　(4) Statuscode

二、选择题

(1) A　(2) B　(3) D

第 11 章

一、填空题

(1) 层级选择器　(2) 标签选择器　(3) $("menu > item")
(4) $(":hidden")　(5) 简单过滤　(6) $("are:hidden")
(7) $("li").eq(3).css("background","#FCF");
(8) $("div").css({"color":"#FFFFFF","background-color":"balck"});

二、选择题

(1) B　(2) D　(3) A　(4) A　(5) C　(6) A　(7) D

第 12 章

一、填空题

(1) XMLHttpRequest (2) 服务器 (3) $.get( ) (4) $.ajax( ) (5) $.getJSON( )

二、选择题

(1) C (2) A (3) C (4) D (5) A

第 13 章

一、填空题

(1) Object (2) Setup (3) Request

二、选择题

(1) C (2) A (3) D (4) A